Crestron **SIMPL+**®
Software
Programming Guide

CRESTRON

This document was prepared and written by the Technical Documentation department at:

Crestron Electronics, Inc.
15 Volvo Drive
Rockleigh, NJ 07647
1-888-CRESTRON

Contents

SIMPL+®

Introduction

What is SIMPL+?

SIMPL+ is a language extension to SIMPL Windows®. It does not replace SIMPL, but instead it enhances it. With SIMPL+ it is now possible to use a procedural "C-like" language to code elements of the program that were difficult, or impossible with SIMPL alone.

A SIMPL+ program is a module that directly interacts with the control system. In order to interact with the control system, a module must contain a few essential elements. The first element is a starting point. A starting point is needed for two reasons. First, it serves as a convenient place to initialize any global variables that are declared within the module. Second, any functionality that the module needs to perform on its own (instead of being triggered though an event), can be instantiated here. Another element is event processing. In order for a SIMPL+ module and a control system to interact, they must be able to send and receive signals to and from one another. Input and output (I/O) signals are declared within the module and are then tied directly to the control system. Input signals are sent from the control system and are received within the SIMPL+ module. Output signals are sent from the SIMPL+ module to the control system. Events are functions that are triggered through input signals from the control system. I/O signals can be either digital, analog or serial and are declared within the SIMPL+ module. Events tell the SIMPL+ module that something has changed within the control system and allows the module to perform any action accordingly.

For Whom is this Guide Intended?

This manual assumes the reader has at least a working knowledge of the SIMPL™ Windows® programming environment. This includes the ability to configure a new program (define the hardware), and interconnect user-interfaces (e.g., a touchpanel) and system outputs (e.g., a relay). Knowledge of the SIMPL logic symbols is not required, but is helpful in understanding some of the examples presented herein.

This guide is intended to be the complete SIMPL+ programming guide, appropriate for the beginning SIMPL+ programmer, or the expert programmer looking for a refresher course. This guide, along with *The SIMPL+ Reference Manual* should provide all the information needed for any SIMPL+ programmer.

Using SIMPL vs. SIMPL+

SIMPL+, while exciting and powerful, does present the programmer with somewhat of a dilemma, namely, when to program in SIMPL and when in SIMPL+. The answer of course is not cut-and-dry, and just about any task can be accomplished entirely in one language or the other. However, the true power of Crestron® control system programming is unleashed when the strengths of both environments are harnessed simultaneously.

First, almost every program to be written will have some elements of SIMPL. Any time a button is needed to act as a toggle, or it is necessary to interlock a group of source buttons, it is generally simpler to handle these tasks with SIMPL.

SIMPL+ is advantageous for more complex and algorithmic tasks, such as building complex strings, calculating checksums, or parsing data coming from another device. In addition, complex decision-making, especially when dealing with time and date, is generally much easier to handle in SIMPL+. Finally, data storage and manipulation may be better suited to SIMPL+ than to SIMPL (though many SIMPL programs have been written to do these chores).

Of course, ultimately the decision as to how to program is up to the individual. Personal preference certainly comes in to play. With practice, a happy medium can be found that makes programming both efficient and fun.

What is Needed to Use *SIMPL+*?

SIMPL+ version 2.0 requires a CNX-series control processor and SIMPL Windows v1.23 or later.

SIMPL+ version 3.0 accompanies SIMPL Windows v2.00 or later, and may be used to program either a 2-Series control system or a CNX-series control system.

Where Can I Get More Information?

This guide should contain all the information needed to program in SIMPL+. For specific information about the language syntax, refer to the latest revision of the SIMPL+ Language Reference Guide (Doc. 5797).

Quick Start

Writing Your First *SIMPL+* Program: "Hello world!"

The best way to become acquainted with SIMPL+ is to write a simple program right off the bat. Although programs can be written in SIMPL+, it is important to understand that all control system "i/o" must be defined directly in SIMPL Windows. This SIMPL Windows program can be thought of as a "shell" in which the SIMPL+ modules are contained. This shell consists of hardware definitions at the very least, but in most cases also consists of raw SIMPL code. SIMPL+ program(s) appear as logic symbols in the overall SIMPL program.

Based on the fact that SIMPL+ programs can exist only inside this wrapper, it is necessary to create a skeleton SIMPL Windows program before testing the program. This is covered in a later section (to be supplied). For now, concentrate on writing the SIMPL+ code only.

Start creating a new SIMPL+ program while running SIMPL Windows. Select **File | New | New SIMPL+ Module**. The SIMPL+ programming environment appears. Instead of a blank window, a skeleton program filled with commented code shows up. This commented out code makes it easy to remember the language syntax and structure. Simply locate the necessary lines, uncomment them, and add the appropriate code. To uncomment a line of code, either remove the "//" that appears at the start of the line or remove the multi-line comment indicators /*…*/.

SIMPL+ programs communicate with the SIMPL Windows wrapper program via inputs and outputs. These inputs and outputs correspond to signals in the world of SIMPL and can be digital, analog, or serial signals (if these terms are unfamiliar, they are covered in more detail in "Input/Output Types"). For this first program, only a single digital input is defined. Find the line of code that says "// DIGITAL_INPUT". Uncomment it and edit it so it looks like the following:

```
DIGITAL_INPUT speak;
```

This line defines the variable speak as the first digital input to the SIMPL+ program. Notice that most lines in SIMPL+ end in a semi-colon (;). To be precise, all statements end with a semi-colon. The definition of a statement in SIMPL+ can be found in the latest revision of the SIMPL+ Language Reference Guide (Doc. 5797).

When a digital input goes from low to high, a push event is generated. To define a push event function for that signal, program this function to yield the desired actions. From the skeleton program, find the commented line of code that says "push {". Uncomment the function block by removing the surrounding comment characters and edit it to read the following:

```
PUSH speak
{
    Print( "Hello world!"\n );
}
```

This function causes the string *"hello world"* plus a carriage return and line feed to be sent out the control system computer port (or Ethernet port) whenever the signal speak goes high. Notice the curly-braces ({}) surrounding the print statement above. In SIMPL+ these braces are used to group multiple statements into a compound statement. In the case of a function definition, always surround the contents of the function with these braces.

The next step is to add another event function, one that responds when a signal goes from high to low. This event is called a release event. From the skeleton program, find the line of code that says "RELEASE input". Uncomment and edit it to read the following:

```
RELEASE speak
{
    Print( "Crestron people make the difference"\n );
}
```

Finally, define what happens when the control system first boots up. This is accomplished using *Function Main*. Upon system startup, the program code defined in this function executes. Unless there are looping constructs (discussed in "Controlling Program Flow: Loops") defined in this function, this code executes only one time for the life of the control system (or until it is rebooted). From the skeleton program, find the section of the program that says "Function Main". Edit it to read the following.

```
Function Main
{
    Print( "I am born!"\n );
}
```

This causes the text "I am born" to be sent out the computer port only upon startup.

To save the file, from the menu, select **File | Save**. Assign the name, *My first SIMPL+*. To compile the file, select **Build | Save and Compile**. This command saves the code module, compiles it, and tells SIMPL Windows how to present it to the SIMPL programmer. SIMPL+ version 2.0 requires that all SIMPL+ modules reside in the User SIMPL+ directory (this can be checked in SIMPL Windows by selecting **Edit** | **Preferences** and clicking on the *Directories* tab). In SIMPL+ 3.0 and later, SIMPL+ modules can also reside in the corresponding SIMPL Windows Project Directory, where the SIMPL Windows program also resides.

Each time the program is saved, an "update log" appears at the bottom of the screen. This log shows the results of the save, compile, and update process that just occurred. Review and become familiar with it. The window should display something similar to this code:

```
Compiling c:\Crestron\simpl\usrsplus\my first simpl+.usp
Total Error(s): 0
Total Warning(s): 0
SIMPL+ file saved successfully
No errors found: SIMPL Windows Symbol Definition updated
```

This first SIMPL+ program is complete. The next section explains how to incorporate this program into the required SIMPL Windows wrapper, and how to run and test it.

Making it Work

This section describes how to make the simple SIMPL+ program written in the last section work inside a Crestron control processor. As was mentioned earlier, SIMPL+ programs cannot run all by themselves. They must be enclosed inside a SIMPL wrapper. This section discusses how to set up this program in SIMPL Windows.

Create a new SIMPL Windows program and add a control processor from the *Configuration Manager*. Notice that only CNX-series or 2-Series control processors are compatible with SIMPL+. For this example, use the *Test Manager* to trigger the digital input. As a result, there is no need to define a touchpanel or other user-interface device, although it is even better if one is available for testing.

After system is configured, switch to the *Program Manager* and make sure that the symbol library pane is visible on the left-hand side of the screen. Find the *User SIMPL+* folder and open it. An icon representing the SIMPL+ program written in the previous section appears. Drag this icon into the *Logic* folder in the *Program View* pane. The SIMPL+ program now becomes just another symbol in the program.

Double click on the logic symbol to bring it into the *Detail* window. It should have a single input, labeled "speak." This of course corresponds directly to the declarations section of our SIMPL+ code, where only a single input and no outputs were defined. Define a signal for this input. The signal name here is not important, but for this example, call it "test_me." Also note that if a user interface was defined in an earlier step, assign this same signal to a button press.

That's it! The first program is complete. All that is left is to compile the whole thing, transfer it to the control processor, and test it. As in SIMPL Windows, compile the

program by clicking on the compile toolbar button or selecting **Project | Convert/Compile**. The compile process automatically recognizes that there is a SIMPL+ module in the program and compiles it along with the SIMPL code (even though it was already compiled when it was saved; SIMPL Windows always recompiles because it must link the modules together with the SIMPL Windows program).

After compilation, transfer the program when prompted by SIMPL Windows. Click **YES** and the SIMPL code section is sent first (followed by the save permanent memory image). Once completed, the SIMPL+ program code is sent. The SIMPL+ code resides in a separate area of memory, thus it is transferred in a separate step. Also realize that the SIMPL+ program is written directly to flash memory, thus there is no need to create a permanent memory image.

At this point a program has been loaded into the control system and is ready to be tested. Start *Test Manager*, and select **Status Window | Add Signal**. A list of all the signals defined in the program appears. Click on the "test_me" signal and then click the **Add** button. Then select **Close**. An icon representing the signal in the status window appears.

The program is ready to be tested. Make sure that the section of *Test Manager* labeled *Incoming Data* is visible in the left-most pane. Click on the "test_me" icon in the status window and then on the **Assert** button on the toolbar (or select **Status Window | Assert Signals**). The signal is driven to the high state, which triggers the push event. In the *Incoming Data* window, the string "Hello world!" appears.

Click on the **De-Assert** button to drive the signal low and trigger the release event. In the *Incoming Data* window, the string "Goodbye cruel world!" appears.

By clicking on the **Positive Pulse** button, both strings appear one after the other, since the push and release events are triggered in rapid succession.

Finally, what happened to the startup text "I am born"? Remember that Function Main only runs on system startup and this occurred even before *Test Manager was* started. Thus it was missed. To see it now, reboot the control processor by selecting **Options | Reset Rack**.

In addition to the latest revision of the SIMPL+ Language Reference Guide (Doc. 5797), continue reading through this manual to learn more about how to program in SIMPL+.

The Structure of a *SIMPL+* Program

What are the different elements that make up a SIMPL+ program? This section provides an overview of the code structure, given in the typical order that they are used.

Compiler Directives

Compiler directives should come at the beginning of the program, and are used to provide explicit instructions to the compiler. As such, these elements are not part of the SIMPL+ language itself. These directives are distinguished from actual SIMPL+ code by preceding them with a pound sign (#).

Currently there are seven compiler directives, each of which is provided in the template file that is created when a new program is started. The compiler directives are as follows.

#SYMBOL_NAME - Allows the user to specify the name that SIMPL Windows uses for this module. If this directive is left out, the filename will be used by default.

#HINT - Provides text that appears in the SIMPL Windows status bar whenever the module icon is clicked on.

#CATEGORY – (SIMPL+ 3.0 and later) Specifies the SIMPL Windows symbol tree category number for this SIMPL+ module, which controls where the SIMPL+ module is listed in the symbol tree in Program Manager. Selecting **Edit | Insert Category** from the menu will display a list of available categories to choose from and automatically insert the selected category in to the program module.

#DEFAULT_VOLATILE – (SIMPL+ 3.0 and later) Specifies that all program variables will retain their values if hardware power is lost. If neither the #DEFAULT_VOLATILE nor #DEFAULT_NONVOLATILE are specified, the compiler will default all variables declared within the SIMPL+ module as nonvolatile.

#DEFAULT_VOLATILE – (SIMPL+ 3.0 and later) Program variables will not retain their value if hardware power is lost.

#HELP_BEGIN / #HELP_END - Allows on-line help to be entered for this module. This text appears when the user selects the module and presses F1 from within SIMPL Windows.

#DEFINE_CONSTANT - Allows constant numeric/string values to be assigned to alphanumeric names. This is extremely useful for writing changeable and readable code.

This last compiler directive deserves more discussion, since using constant definitions are a very important part of writing readable code. To illustrate this, examine the following example.

```
PUSH vcr_select
{
    switcher_input = 3;
    switcher_output = 2; // video projector
}

PUSH dvd_select
{
    switcher_input = 4;
    switcher_output = 2; // video projector
}
```

In this example it should be clear that the value of a variable, *switcher_input*, is being set to 3 if the vcr button is pressed or 4 if the dvd button is pressed. In both cases, the variable, *switcher_output*, is set to 2, which is the output connected to the video projector. Presumably, these variables would be used somewhere else in the program to generate a command string to control a switcher. Using numbers in a small and simple program like this still produces a relatively readable program. Even so, a couple of problems should become evident. For one thing, if the switcher configuration is changed, and the inputs and outputs are rearranged, the user must carefully go through the program and change all the appropriate values for the switcher input and output. Secondly, in a larger program this technique becomes very hard to read. After all, the number 3 has no intrinsic relationship to a VCR.

Examine the following equivalent program, which uses constant definitions in place of actual numbers.

```
#DEFINE_CONSTANT VCR_INPUT 3
#DEFINE_CONSTANT DVD_INPUT 4
#DEFINE_CONSTANT VPROJ_OUTPUT 2


PUSH vcr_select
{
    switcher_input = VCR_INPUT;
    switcher_output = VPROJ_OUTPUT; // video projector
}


PUSH dvd_select
{
    switcher_input = DVD_INPUT;
    switcher_output = VPROJ_OUTPUT; // video projector
}
```

Note the use of capital letters for the constant definitions. This is not required, but it makes it clear to see the difference between variables and constants when reading through a program (but of course is not useful if all caps are used for the rest of the program). Not only is this version of the program easier to read, even for a small example, but it is obvious that changing a numeric value in one place (the #DEFINE_CONSTANT) can affect the value everywhere in the program.

Include Libraries

Libraries are a way of grouping common functions into one source file to enable modularity and reusability of source code. Libraries are different from modules in that they do not contain a starting point (*Function Main*), and cannot interact with the control system (through I/O signals and events). Libraries can include other libraries, but cannot include a SIMPL+ module. Only functions and defined constants are allowed to be declared and defined within libraries. Global variable declarations are not allowed. Functions, however, can contain local variables. Other advantages are:

1. Modularity. SIMPL+ programs can grow to be large and can be better organized by taking sections of code and placing them into a User-Library. It is best to create libraries that contain sets of related functions. For example, a library might be created that contains only functions that perform certain math related functions. Another library might be created that contains functions performing special string parsing routines.
2. Reusability. As modules are written, it is common for SIMPL+ modules to need pieces of functionality that were previously written in other modules. These common and repeatedly portions of code can be extracted and placed into one or more libraries. Once placed into a library, one or more SIMPL+ modules can include and make use of them.

SIMPL+ modules include libraries using the following syntax:

```
#USER_LIBRARY "<library_name>"
#CRESTRON_LIBRARY "<library_name>"
```

Note that *library_name* is the name of the library **without** the file extension. User-Libraries are libraries that the end user writes. These can exist either in the SIMPL+ module's project directory, or the User SIMPL+ directory (set in SIMPL Windows).

Crestron-Libraries are provided from Crestron and are contained within the Crestron Database.

Variable Declarations

Variables can be thought of as storage areas to keep data. When writing all but the most basic of programs, users need to use variables to store values.

Any variable used in a SIMPL+ program must be declared before it is used. This also tells the operating system how much space must be reserved to hold the values of these variables. This section describes the different types of variables in SIMPL+ and how to define them.

Inputs, Outputs, and Parameters

The current release of SIMPL+ does not support passing parameters (constant values) from the SIMPL program into the SIMPL+ module. Look for this feature in a future release.

SIMPL+ programs communicate with the SIMPL program in which they are placed through input and output variables and through parameter values. This is similar in concept to the Define Arguments symbol used in SIMPL macros. Input variables can be of three types: digital, analog, and string types. These correspond directly to the same signal types in SIMPL and the buffer input, which is a special case of the string input. Output variables can only be of the digital, analog, or string variety.

Input variables are declared using the following syntax.

For more information on the buffer_input, refer to "Working with Strings" on page 45.

```
DIGITAL_INPUT <dinput1>,<dinput2>,…<dinputn>;
ANALOG_INPUT <ainput1>,<ainput2>,…<ainputn>;
STRING_INPUT <sinput1>[size],<sinput2>[size],
...<sinputn>[size];
BUFFER_INPUT <binput1>[size],<binput2>[size],
...<binputn>[size];
```

Digital and analog output variables are declared in the same way, except the word input is replaced with output, as follow shown below. String output variables do not include a size value. There is no output version of the buffer variable.

```
DIGITAL_OUTPUT <doutput1>,<doutput2>,…<doutputn>;
ANALOG_OUTPUT <aoutput1>,<aoutput2>,…<aoutputn>;
STRING_OUTPUT <soutput1>,<soutput2>,…<soutputn>;
```

The inputs and outputs declared in this way govern the appearance of the SIMPL+ symbols that are presented via SIMPL Windows. The order of the signal declarations is important only within signal types; in SIMPL Windows, digital signals always appear at the top of the list, followed by analogs, and then serials.

Variables

In addition to the input and output variables described in the last section, the user can define and use variables that are only seen by the SIMPL+ program. That is, the SIMPL program, which holds this module has no knowledge of these variables. In addition, any other SIMPL+ modules that are included in the SIMPL program would not have access to these variables.

"Working with Data (Variables)" discusses variables in much more detail. For now, understand how to declare them. Declaring variables tells the SIMPL+ compiler how much memory to put aside to hold the workable data.

These variable declarations are very similar to input/output declarations. However, instead of digital, analog, and serial (string and buffer) types, integer and string

variables are also available with the INTEGER and STRING datatype. Integers are 16 bit quantities. For the 2-series control system, 32 bit quantities are supported with the LONG_INTEGER datatype. Both INTEGER and LONG_INTEGER are treated as unsigned values. Signed versions for both of these datatypes are available by using the SIGNED_INTEGER and SIGNED_LONG_INTEGER datatypes. The following example illustrates how each of these datatypes can be used within a program module:

```
INTEGER intA, intB, intC;
STRING stringA[10], stringB[20];
LONG_INTEGER longintA, longIntB;
SIGNED_INTEGER sintA, sintB;
SIGNED_LONG_INTEGER slongIntA;
```

It is important to realize that all variables declared in this manner are non-volatile. That is, they remember their values when the control system reinitializes or even if the power is shut off and then turned back on. Since input/output variables are attached directly to signals defined in the SIMPL program, they do not have this property unless the signals they are connected to are explicitly made non-volatile through the use of special symbols.

Structures

Sometimes sets of data are needed rather than individual pieces. Variables store a piece of data, but are not related to other variables in any way. Structures are used to group individual pieces of data together to form a related set. Before structures can be used, a structure definition must be defined. Defining a structure is really defining a custom datatype (such as STRINGs and INTEGERs). Once this new type (the STRUCTURE) is defined, variables of that type can be declared. The following example illustrates how a structure can be defined and used within a program module:

```
STRUCTURE PhoneBookEntry
{
   STRING name[100];
   STRING address[100];
   STRING phone_number[25];
   INTEGER age;
}

PhoneBookEntry entry;
PhoneBookEntry entries[50];
```

To access a variable within a structure, the structure's declared variable name is used, followed by a period (also known as the 'dot' or 'dot operator'), followed by the structure member variable name. For example:

```
entry.name = "David";
entries[1].age = 32;
```

User-Defined Functions

The term checksum byte is commonly used in serial communications to represent a byte (or bytes) that is appended to a command string. This byte is calculated from the other characters in the string using some specified algorithm. Checksum bytes are used to provide error-checking when communicating between devices.

In programming, it is common to reuse the same code over and over again. For example, when writing a program to generate strings (to control a device), there may be a need to calculate a checksum byte. Once the code to calculate this byte is formulated, paste it in to the program after each instance where a command string is created.

This technique has many flaws. First, the program can grow unnecessarily large and become hard to manage and debug. Second, if there is a need to change the code, it must be changed every place it was used, which is time consuming and error prone.

The solution is to create user-defined functions to perform common tasks. A user-defined function is very similar to a "built-in" function like Date or MakeString, with some important exceptions.

To invoke a user-defined function, use the following syntax:

```
CALL MyUserFunction();
```

Event Functions

Event functions make up the heart of most SIMPL+ programs. Since a well-designed control system is "event-driven" in nature, most code is activated in response to certain events when they occur. Event functions allow the user to execute code in response to some change that has occurred to one or more of the input signals feeding the SIMPL+ module from the SIMPL program.

Two things must be realized about event functions. They can be used with input variables only (not with locally defined variables). Also, they are only triggered by the operating system at the appropriate time (that is, they cannot be called manually by the programmer).

Like everything else in the control system, event functions are multi-tasking. That is, an event can be triggered even if another event in the same SIMPL+ module is already processing. As described in "Understanding Processing Order" on page 49, this only happens if events are triggered on the same logic wave, or if one event function has caused a task switch.

The structure of an event function is as follows.

```
event_type <input list>
{
    <statements>
}
```

In SIMPL+ there are three basic event types that can occur: PUSH, RELEASE, and CHANGE. In addition to these three is a fourth type simply called "EVENT." These event types are discussed in the following subsections.

PUSH and RELEASE Events

Push and release events are valid only for DIGITAL_INPUT variables. The push event is triggered when the corresponding digital input goes from a low to a high state (positive- or rising-edge). The release event occurs when the signal goes from a high to a low (negative- or falling-edge). For example, the following code sends a string to a camera unit to pan left when the left button is pressed and then send a stop command when the button is released.

This example assumes that the camera unit being controlled continues to move in a given direction until a stop command is issued. Some devices function this way, but others do not.

```
DIGITAL_INPUT cam_up, cam_down, cam_left, cam_right;
STRING_OUTPUT camera_command;


PUSH cam_left
{
    camera_command = "MOVE LEFT";
}


RELEASE cam_left
{
    camera_command = "STOP";
}
```

CHANGE Events

Change events can be triggered by digital, analog, string, or buffer inputs. Anytime the corresponding signal changes its value, the change event will be triggered. For digital signals, this means that the event will trigger on both the rising and falling edges (push and release). For buffer inputs, this event triggers any time another character is added to the buffer.

The following example sends a command to a CD player to switch to a different disc whenever the analog input disc_number changes value.

```
ANALOG_INPUT disc_number;
STRING_OUTPUT CD_command;


CHANGE disc_number
{
    CD_command = "GOTO DISC " + itoa(disc_number);
}
```

This program uses the *itoa* function to convert the analog value in disc_number into a string value which can be concatenated onto CD_command. The string concatenation operator (+) and system functions (i.e., *itoa*) are discussed in later sections of the manual and in the latest revision of the SIMPL+ Language Reference Guide (Doc. 5797).

Compound Events

Sometimes it is desired to have the same (or similar) action occur when any of a number of events occur. For example, there may be a need to generate a switcher command string each time any of a group of "output" buttons are pressed.

Compound events can be created in two ways. One way is to provide a list of input signals separated by commas in the event function declaration. Refer to the following example.

```
PUSH button1, button2, button3
{
    <statements>
}
```

A second form of compound event occurs when combining different types of events into a single function. For example, there may be a need to execute some code when a button is pushed or the value of an analog signal changes. To accomplish this, stack the event function declarations, as follows.

```
CHANGE output_value
PUSH button1, button2
{
   <statements>
}
```

A useful feature of SIMPL+ event functions is that a single input can have more than one event function defined for it. This makes it possible to write one event function for a specific input only and another event function for a group of inputs. Refer to the following example.

```
PUSH button1
{  // code here only runs when
   // button1 goes high
}


PUSH button1, button2, button3
{  // this code runs when any of
   // these inputs goes high
}
```

The Global Event

A special form of event exists, which is triggered anytime any of the inputs to the SIMPL+ module changes. This is simply a shortcut for having to build a compound event manually which includes all the inputs separated by commas in a CHANGE event declaration. Access this special event function by using the EVENT keyword, as follows.

```
EVENT
{  // this code runs anytime anything
   // on the input list changes
}
```

Be careful when using this global event function. If the user has a SIMPL+ program in which a change on any input causes the same code to execute, this type of event is useful. However, if additional inputs are added at a later time, remember that this event function exists, and is caused when these new inputs change as well. This may not be desirable.

Function Main

Main is a special case of a user-defined function. The Main function is executed when the control system initializes (boots up) and is never called again. In many cases, the main function is used to initialize variables; it may not contain any statements at all.

However, in some cases, a loop may be placed in the Main function to perform a continuous action. Refer to the following example, and note that this example uses a while loop construct, which is discussed in “Controlling Program Flow: Loops” on page 27.

```
Function Main()
{
   x = 0;    // this code executes only once
```

```
    while (1)
    {
       <do something forever>     // code in here runs
                                  // continuously
    }
}
```

This loop runs continuously for as long as the control system is on. If a construct like this is used, it is recommended that a *ProcessLogic* or *Delay* function in the loop be included to allow the logic processor a chance to handle the rest of the system. If one of these statements is not included, the operating system forces a task switch at some point in time. These concepts are discussed in detail in "Understanding Processing Order" on page 49.

Working with Data (Variables)

Programming is really the manipulation of data. Examples of data in a program are the switcher input and output numbers, the name of the next speaker and the amount of time left before the system shuts down automatically. This section covers the different data types available in SIMPL+.

Input/Output Types

Input/output variables are used to transfer data between SIMPL+ modules and the surrounding SIMPL program. Each input or output variable in the SIMPL+ is connected directly to a signal in the SIMPL program. SIMPL programmers should already be familiar with the three signal types available in that language: digital, analog, and serial. The table below takes a closer look at the type of data conveyed by these signal types.

SIMPL Signal Types

SIGNAL TYPE	DATA	EXAMPLE
Digital	Single bit	Button push/release
Analog	16-bit (0 to 65,535)	Volume level
Serial	Up to 255 bytes	Serial data input from a COM port

This table illustrates that digital signals only transfer a single bit of information between SIMPL+ and SIMPL. Of course this makes sense, as digital signals only have two possible states (on and off). Obviously, analog and serial signals allow the transfer of much more information per signal. Depending on the application, it may be more convenient to generate an analog signal in SIMPL and connect it to a SIMPL+ program, rather than connecting a large number of digital signals and setting some variable based on which was pressed last (though both methods should work).

Digital Inputs/Outputs

Digital signals comprise the bulk of signals in a typical SIMPL program. In SIMPL+ they are used mainly to trigger events on the rising- or falling- edge of the signal, though they can also be used in expressions.

The state (or value) of a digital signal is always either 1 or 0 (also referred to as ON or OFF). In SIMPL+, assigning a value of 0 to a digital signal turns it OFF. Assigning it any non-zero value will turn it ON (for clarity, in most cases, use the value 1).

Analog Inputs/Outputs

Analog signals are used in SIMPL to accomplish tasks for which digital signals are inadequate. Typical examples include volume control and camera pan/tilt control. In SIMPL+, analog signals take on even greater importance since they provide an easy way of transferring data (16 bits at a time) into and out of SIMPL+ modules.

In SIMPL+, analog signals are treated much as they are in SIMPL. They are 16-bit numbers that can range between 0 and 65,535 (unsigned) or –32768 and +32,767 (signed). Signed and unsigned numbers are discussed in detail in "Integers" on page 16.

String Inputs/Outputs and Buffer Inputs

Perhaps the greatest advantage that SIMPL+ provides is related to string handling. In SIMPL, serial data can be generated dynamically and placed on serial signals. However, one problem that arises is due to the "transient nature" of these signals. Simply put, serial signals are invalid except for the time between when they are created and when they reach the next symbol. With careful programming this does not cause problems, but it requires a good understanding of SIMPL.

SIMPL+ makes working with serial data much simpler by storing this data into temporary string variables. When a serial signal is connected to a SIMPL+ module and the SIMPL program causes data to be sent via this signal, the SIMPL+ program copies the data from this signal into local memory. The data is kept there until the SIMPL program changes it.

By storing the serial data into a string variable, SIMPL+ programmers can now perform tasks on strings that were difficult or impossible with SIMPL alone. For example, it is easy to evaluate a string and then add a checksum byte on the end to insert or remove characters from a string, or to parse information out of a string for use elsewhere. Functions that are designed to work explicitly with string signals and string variables are discussed in detail in the latest revision of the SIMPL+ Language Reference Guide (Doc. 5797).

Serial data is also unique in that unlike digital or analog signals, the data may not appear at one time, but instead it can "stream in" (e.g., if it comes from a COM port). This raises an interesting problem, namely, what happens if a command string coming in from a device is not picked up as one piece, but rather is broken up into two or more pieces? The problem arises in that a string input is completely replaced each time new data is detected on the input. To account for this, an alternate type of serial input type may be used, the buffer input. The buffer input differs from the string input in that serial data that comes in is appended onto data that already exists in the buffer, instead of replacing it. This type of behavior is critical for performing sophisticated string parsing and manipulation when dealing with streaming data. Refer to "Working with Strings" on page 45 for a detailed discussion of buffer inputs.

Signal Scope

Signals are global throughout the entire SIMPL program and to any SIMPL+ program to which they are connected. In a SIMPL+ program, the values of digital, analog, and string inputs are read (their values can be evaluated). However, their values cannot be changed from within the SIMPL+, thus they are considered read-only. Buffer inputs can be read from and modified.

Digital and analog output signals in SIMPL+ can be read and modified. String outputs can be modified, but cannot be read back. To understand this, the user must realize what is being seen when looking at the contents of an output variable in a

SIMPL+ program. The value of any output is the value of the signal as seen by the outside SIMPL program at that instant. This is critical considering that SIMPL+ does not necessarily "propagate" outputs to the SIMPL program each time they are changed in the program. As a general rule, assume that analog and serial outputs are propagated at the time they are assigned new values. However, digital signals are not propagated until a task switch occurs.

This explains reading values of analog and digital outputs, but why is it that string outputs cannot be read? The reason has to do with the nature of serial signals in SIMPL. Namely that these signals do not actually store strings in them, but rather point to locations in memory where a string exists. Since the data stored at a particular location in memory can change at some later time, there is no guarantee that the string data is still there. As a result, SIMPL+ does not allow a string output to be examined.

Examine the following code example.

```
DIGITAL_OUTPUT d_out;
ANALOG_OUTPUT  a_out;
STRING_OUTPUT s_out;


PUSH someEvent
{
    d_out = 1; // set this digital output to 'on'
    a_out = 2000; // set this analog output to 2000
    s_out = "hello"; // set this string output to "hello"


    if (d_out = 1)  // this WILL NOT be true until the
        Print ("d_out is on\n");  // next task-switch


    if (a_out = 2000)       // this WILL be true
        Print ("a_out = 2000");


    if (s_out = "hello")    // this WILL NOT be true due to the
        Print ("s_out is hello"); // nature of serial signals


    ProcessLogic();  // force a task-switch


    if (d_out = 1)  // NOW this is true
        Print ("d_out is on\n");
}


Function Main()     // initialization
{
    d_out=0;
    a_out = 0;
}
```

The 'if' language construct is described in detail in "Controlling Program Flow: Branching". Evaluation of TRUE and FALSE expressions are covered in "Operators, Expressions, and Statements" on page 22.

In this example, the digital output, *d_out*, and the analog output, *a_out*, are set to 0 on system startup in the *Function Main*. In the push function, the first conditional *if* statement evaluates to FALSE because the digital output signal, *d_out*, is considered OFF until this value is propagated to the SIMPL program. With digital outputs, this does not happen until the SIMPL+ program performs a task switch. The analog and string outputs, on the other hand, are propagated as soon as they are assigned new values. Thus the second *if* condition evaluates to TRUE and the subsequent print statement is executed. The third *if* statement can still evaluate to FALSE however, due to the nature of serial signals in SIMPL, as previously described.

Notice the *ProcessLogic* function call in the last example. This function forces a task switch from SIMPL+ to the SIMPL logic processor. This causes the digital signal to be propagated out to the SIMPL program. The next time the logic processor passes control back to this SIMPL+ program, it picks up where it left off. As a result, the fourth *if* condition evaluates to TRUE, thus executing the print statement.

All About Variables

In addition to input and output signals, additional variables can be declared that are only used inside the SIMPL+ program. That is, the outside SIMPL program has no knowledge of these variables and no access to them. These variables are critical for use as temporary storage locations for calculations.

Unless otherwise specified with a compiler directive, all variables in SIMPL+ are "non-volatile," which means that they remember their values even after the control system is shut off. This can be extremely useful, though it does require some caution. In the 2-series control system, the compiler directive, #DEFAULT_VOLATILE, can be used to change this behavior so that the variable's values are not retained after the control system is shut off. Notably, it is generally a good idea to explicitly initialize variables to some value before using them, except in the cases where it becomes necessary to take advantage of their non-volatility. An obvious place to do this is in *Function Main*.

SIMPL+ allows for two different types of variables: integers and strings. In addition, variables of either type may be declared as one- or two-dimensional arrays. The following sections explain these topics in detail.

Integers

Integers contain 16-bit "whole numbers." That is, they can range between 0 and 65,535 (unsigned, refer to paragraph after example) and cannot contain a decimal point. SIMPL programmers may recognize that this range is identical to that of analog signals. This is because analog signals are also treated as 16-bit values.

Integers are declared as follows:

```
INTEGER <int1>, <int2>,…,<intn>;
```

Depending on how they are used, integers can either be "unsigned" or "signed." Unsigned integers have values between 0 and 65,535. Signed integers have values between –32768 and +32767.

In reality, there is no difference between a signed and unsigned integer, the difference is solely in how the control system views them. That is, for any given value, that number can be thought of as either being a signed number or an unsigned number. Depending upon which operations are perform on a number, the control system decides whether to treat that number as signed or unsigned.

When an integer has a value of between 0 and 32767, it is identical whether it is considered signed or unsigned. However, numbers above 32767 may be treated as negative numbers. If they are, they will have a value of x – 65536, where x is the unsigned value of the number. This means that the value 65,535 has a signed value of –1, 65534 is –2, etc. This scheme is referred to as two's complement notation.

Why is all this signed/unsigned nonsense important? Well, in most cases, it can be ignored and things work out fine. However, for those instances when it does make a difference, it pays to understand how to debug programs that are not working as expected.

In control system programming, often there is not a need for negative numbers (e.g., how often is a switcher switched to input number –12 ?). As a result, the most common operations treat integers as unsigned and it becomes necessary to use special "signed" operators or functions when treating numbers as signed. The table after this paragraph lists operators and functions that are unsigned and those that are signed. Any operators or functions that are not shown here do not need special consideration.

Unsigned/Signed Operators and Functions

DESCRIPTION	UNSIGNED OPERATORS/FUNCTIONS	SIGNED OPERATORS/FUNCTIONS
Less than	<	S<
Less than or equal to	<=	S<=
Greater than	>	S>
Greater than or equal to	>=	S>=
Integer division	/	S/
Maximum	Max()	SMax()
Minimum	Min()	SMin()

Examine the following:

```
INTEGER j, k;

Function Main()
{
    j = 2;
    k = -1;  // this is the same as k = 65535


    if (j > k) // this will evaluate to FALSE
        Print( "j is bigger as unsigned numbers\n" );


    if (j S> k) // this will evaluate to TRUE
        Print( "j is bigger as signed numbers\n" );
}
```

In this example, the first condition, *j* > *k*, evaluates to FALSE and the *Print* statement does not execute. This is because the > operator performs an "unsigned greater than" operation. If both *j* and *k* are converted to unsigned values, *j* remains at 2, but *k* becomes 65,535, and thus *k* is obviously not smaller than *j*.

The second condition, *j S*> *k*, evaluates to TRUE, because this time the "signed greater than" operator was used.

Examine one more example:

```
INTEGER a, b, c, d;

Function Main()
{
    a = 100;
    b = -4;

    c = a / b;  // c = 0 (100/65532)
    d = a S/ b; // d = -25 (100/-4)
}
```

Notice that c calculates to zero because the / operator is unsigned. It treats the variable b as +65,532. Since the / operator truncates the decimal portion of the result, c becomes zero. In regard to the variable d, since the signed division operator S/ was used, b is treated as –4 and the result is –25.

A final note regarding signed/unsigned integers, if an operation results in a number that is greater than 65,535 that number "overflows" and the value wraps around again starting at zero. This allows certain operators (e.g. +, -, and *) to operate with no regard to sign (the result is accurate when thinking of the numbers as signed or unsigned). This also means that when trying to add (or multiply) two unsigned numbers and the result is greater than 65,535, the answer may not be what is expected.

Strings

String variables are used to hold multiple characters in a single variable. The term string is used to illustrate the act of "stringing" together a number of characters to produce words, sentences, etc. Typically strings in SIMPL+ are used to hold things such as serial commands, database records, and so on.

Strings are declared as follows:

```
STRING <string1[size]>, <string2[size]>,…,<stringn[size]>;
```

The number in square brackets following the variable name defines the size of the string variable. When declaring strings, choose a size that is large enough to hold any amount of data that might be needed, but that is not overly large so as to waste space. That is, it is unnecessary to set the variable size to 100 characters when a given variable in an application does not contain more than 50 characters.

Working with strings is not unlike working with other variable or signal types. To assign a value to a string, for example, do the following:

```
STRING myString[50];
myString = "Tiptoe, through the tulips\n";
```

In the example above, a variable called *myString* is declared, which can contain up to 50 characters of data. The value, *"Tiptoe, through the tulips\n"*, is the value being assigned to the variable. The double-quotation marks surrounding the data defines a literal string expression. That is, a string expression which is defined at compile-time (when the program is compiled) and cannot change during run-time (while the program is running). Also note the "\n" at the end of this string literal. This represents a "newline," or a carriage return followed by a line feed. This character combination is used often, so a shortcut was developed. For a complete list of similar shortcuts, refer to the latest revision of the SIMPL+ Language reference Guide (Doc. 5797). Finally, note that the square brackets were not included after the variable

name, as was done when it was declared. When assigning a value to a string, that value is always assigned starting at the first character position.

It is important to note that the length of the this value does not exceed total length allocated for *myString* (in this case, 50 characters). If the declared variable is not large enough to hold the data being assigned to it, the data is truncated to as many characters as the string can hold. Also, when the program is being executed, a string overflow error will be reported within the control system's error log.

The example above is useful, but does not really begin to tap the enormous string-generating capabilities of SIMPL+. A common task in control system programming is the control of an audio/video matrix router via RS-232. To control such a router, often it is necessary to specify the input and output which make up the desired matrix crosspoint. As an example, assume the device to be controlled expects to see a command in a format, shown as follows:

```
IN<input#>OUT<output#><CR><LF>
```

This protocol allows the input and output to be specified as numbers. For example, to switch input 4 to output 10, the command would be as follows:

```
IN4OUT10<CR><LF>
```

Where <CR><LF> represents the carriage return and line feed characters. Obviously, for a router of any significant size, the number of possible crosspoints can grow very large. Thus creating a literal string expression for each case would be very inefficient. Instead, build the control string dynamically as the program runs. An easy way to do this is through the use of the "string concatenation" operator (+). Note that this is identical to the addition operator, but the SIMPL+ compiler is smart enough to know whether integers are added or string expressions are concatenated.

This is addressed in the following code:

```
DIGITAL_INPUT do_switch;
STRING_OUTPUT switcher_out[10];
INTEGER input, output;

PUSH do_switch
{
   switcher_out = "IN" + itoa(input) + "OUT" + itoa(output) +
"\n";
}
```

In this example, the + operator is used to concatenate multiple string expressions. The *itoa* function has been used, which converts an integer value (in this case from analog input signals) into a string representation of that number (e.g. 23 becomes "23").

There is an alternate way to build strings in SIMPL+. The *MakeString* function provides functionality similar to the concatenation operator, while providing a bit more power and flexibility. The following line is equivalent to the concatenation statement above:

```
MakeString( switcher_out, "IN%dOUT%d\n", input, output );
```

This syntax is a bit more confusing. The first argument to the *MakeString* function, switcher_out, is the destination string. This is where the resulting string created by *MakeString* is placed. The second argument, the part embedded in double-quotation marks, is the "format specification". This determines the general form of the data.

Notice how the constant parts of the string are entered directly. The interesting thing about the format specification is the %d sequences, which are known as “type specifiers”.

Variable Scope

Variable declarations in a SIMPL+ program can be either global or local. Global variables are defined in the "Define Variables" section of the code, and "exist" throughout the entire SIMPL+ program. This means that any event function or user-defined function can reference and modify global variables. When the value of a global variable is being set or modified, it is reflected throughout the entire program.

Local variables are defined inside a function declaration and exist only inside that particular function. In other words, if a local variable, *byteCount*, were defined inside of a function, *CalcChecksum*, any reference to *byteCount*, outside of the scope of this function (e.g. in another function) will result in a compiler syntax error. Note that different functions can use the same variable names when defining local variables. Take a look at the following example:

```
// simple function to add up all the bytes in a
// string and append the sum as a single byte
// onto the original string.
String_Function CalcChecksum(STRING argData)
{
   INTEGER i, checksum;

   checksum = 0;

   for (i = 1 to len(argData))
      checksum = checksum + byte(argData,i);

   return (argData + chr(checksum));
}
```

In this example, *i* and *checksum* are local variables that only exist inside the function, *CalcChecksum*. This example also introduces an additional way to implement a local variable: by passing it as an argument to the function, as was done with the STRING variable, *argData*. The concept of local variables and argument passing is discussed in detail in the section "User-Defined Functions" on page 31.

While the use of global variables may seem simpler, local variables can help keep your programs better organized and easier to debug. A significant disadvantage of global variables is that you must be careful each time you use or modify a variable that it does not have an adverse effect on another part of the program. Since local variables can only be used inside of a function, this is not a concern.

Arrays

When INTEGER or STRING variables are declared, the user may also declare them as one- or two-dimensional (INTEGERs only) arrays. An array is a group of data of the same type arranged in a table. A one-dimensional array can be thought of as a single row with two or more columns, while a two-dimensional array can be thought of as a table with multiple rows . In SIMPL+, arrays are declared as follows.

```
INTEGER myArray1[15]    // 1-D integer array with 16 elements
INTEGER myArray2[10][3] // 2-D integer array with 11x4 elements
STRING myArray3[50][8]  // 1-D string array with 9 elements
```

The first two examples above define 1D and 2D integer arrays, respectively. The last example looks like it declares a 2D array of strings, yet the comments states that it actually declares a 1D array of strings. Recall that in "Strings", it was necessary to define the maximum size of the string in square brackets, which is the same notation used for arrays. So, in the example above, nine-element array of 50-byte strings is being declared. The user cannot declare a 2D array of strings in SIMPL+.

Another question should have come to mind from the above examples. That is, why does declaring *myArray1[15]* create an array with 16 elements instead of 15? The answer is that array elements start at 0 and go to the declared size (15 in this case). This fact makes for an easy transition to SIMPL+ for programmers of other languages (some of which start at 0 and others which start at 1). That is, if the user is comfortable with treating arrays as starting with element 0, then the user can continue programming in this manner. If however, the user has used languages, which treat the first element in an array as element 1, then the user may want to use that notation instead.

To reference a particular element of an array when programming, use the variable name followed by the desired element in square brackets. Using the arrays declared in the example above, the following statements are all valid in SIMPL+.

```
j = 5;                 // set an integer variable to 5
myArray1[3] = j;       // set the 3rd element of the array to 5
myArray1[j*2] = 100;   // set the 10th element of the array
                       // to 100

myArray2[j][1] = k;    // set the j,1 element of myArray2 to
                       // the value of k

m = myArray2[j][k-1]; // set the variable m to the value in
                       // the j,k-1 element of the array

myArray3[2] = "test"; // set the 3rd element of the string
                       // array to "test"
```

From these examples, it should be clear that the user may use constants, variables, or expressions (discussed in "Operators, Expressions, and Statements" on page 22) inside of the brackets to access individual array elements. Array elements can appear on either side of the assignment (=) operator. That is they can be written to (left side) or read from (right side). Of special interest is the notation used for storing a value into the string array myArray3. Notice that only one set of brackets was used here even though two sets of brackets are needed when declaring the array. Remember that the first set of brackets in the declaration specified the size (in characters) of each string element. Also recall from earlier in this section, that the size field is not included when referring to strings. For example, refer to the following.

```
STRING myString[50];  // declare a 50-character string

myString = "hello!";  // we do not use the size brackets here
                      // to assign a value to a string variable
```

As a result, when working with string arrays, only use one set of brackets, which refer to the array element, not the string size.

Operators, Expressions, and Statements

This section deals with the core programming elements in SIMPL+.

Operators

Operators take one or two "operands" and combine them in some way to produce a result. In SIMPL+ operators can be binary (takes two arguments) or unary (takes a single argument). For example, the + operator is binary (e.g., x + y), while the – operator can be binary or unary (e.g., x – y, or –x are valid). Most operators in SIMPL+ are binary. Notice that operands do not have to be simple constants or variables. Instead they can be complex expressions that result in an integer.

SIMPL+ operators can be classified into three categories: arithmetic, bitwise, and relational. The sections below describe each category briefly. For a complete list of operators and their function, consult the latest revision of the SIMPL+ Language Reference Guide (Doc. 5797).

Arithmetic Operators

Arithmetic operators are used to perform basic mathematical functions on one or two variables. In all but one case, these operations make sense only for integer types (includes analog inputs and outputs). For example, to add two integers, x and y, together, use the addition operator +, as follows.

```
x + y
```

As mentioned in the previous paragraph, use these operators with integers in all but one case. The exception is the + operator when used with string variables. In this case the operator performs a concatenation instead of addition, which is very handy for generating complex strings from smaller parts. This is discussed in more detail later.

Bitwise Operators

Arithmetic operators deal with integers as a whole. Bitwise operators treat the individual binary bits of a number independently. For example, the unary operator *NOT* simply negates each bit in number, while the & operator performs a binary "and" operation to each bit in the arguments (bit0 and-ed with bit0, bit1 with bit1, etc.).

Relational Operators

Relational operators are used in expressions when it is necessary to relate (compare, equate, etc.) two values in some way (the exception to this is the unary operator NOT). When a comparison is done using a relational operator, the result is an integer, which represents TRUE or FALSE. In SIMPL+, TRUE results equal 1 and FALSE results equal 0. In general, any non-zero value is considered by SIMPL+ to be TRUE, while FALSE is always 0.

Typically, relational operators are used to help control the program flow. That is, test certain conditions and the result determines what happens next. This is discussed in more detail in "Controlling Program Flow: Branching" on page 24.

Expressions

As reading through the later parts of this guide, as well as the latest revision of the SIMPL+ Language Reference Guide (Doc. 5797), the term *expression* is mentioned in many places. For example, in describing the syntax of the *if-else* construct, it may be described as the following:

```
if (expression1)
{
   // code to execute
}
else if (expression2)
{
   // code the execute
}
```

In the above example, expression1 and expression2 can be any valid SIMPL+ expression. This section describes what is and what is not an expression.

An expression in SIMPL+ is anything that consists of operators and operands. Operators were discussed previously in this section, and operands are simply the things on which operators act. For example, refer to the following simple expression.

```
x + 5
```

In this expression the operator is the addition operator (+), and the operands are x and 5. Expressions can contain constants, variables, and function calls in addition to operators. One expression may be made up of many smaller expressions. The following are all valid SIMPL+ expressions.

```
max(x,15)
y * x << z
a = 3
(26 + byte(aString,i) mod z = 25
```

Expressions can range from the very simple to the very complex. Also note that the last two expressions contained an equal sign. It is very important to recognize that this operator can have two different meanings based upon where it is used. In the first example above, the equal sign can serve as an assignment operator (assign the value 3 to the variable a) or as an equivalency comparison operator (does the variable a equal 3?). However, an expression cannot contain an assignment (it would then become a statement, discussed in "Statements" on page 24), so it is indeed recognized as a comparison operation. In the second case, the equal sign also serves as a equivalency comparison operator. Here there is no ambiguity since a value cannot be assigned into an expression (as opposed to a variable).

Expressions always evaluate to either an integer or a string. Refer to the following example.

```
x + 5               // this evaluates to an integer
chr(i) + myString   // evaluates to a string
a = 3               // evaluates to 1 if true, 0 if false
b < c               // evaluates to 1 if true, 0 if false
```

The last two expressions are comparisons. Comparison operations always result in a true or false value. In SIMPL+, true expressions result in a value of 1 and false expressions result in a value of 0. Understanding this concept is key to performing decision making in SIMPL+. In reality, any expression that evaluates to a non-zero value is considered TRUE. This concept is discussed in "Controlling Program Flow:

Branching" and "Controlling Program Flow: Loops" on pages 24 and 27, respectively.

Statements

Statements in SIMPL+ consist of function calls, expressions, assignments, or other instructions. Statements can be of two types: simple or complex. Simple statements end in a semicolon (;). Examples of simple statements are as follows:

```
x = MyInt / 10;                    // An assignment
print("hello, world!\n");          // A function call
checksum = atoi(MyString) + 5;     /* Assignment using function
                                      calls and operators */
```

A complex statement is a collection of simple statements surrounded with curly braces ({}). An example of a complex statement would be as follows:

```
{ // start of a complex statement
   x = MyInt / 10;
   print("hello, world!\n");
   checksum = atoi(MyString) + 5;
} // end of a complex statement
```

Controlling Program Flow: Branching

In any substantial program, making decisions must control the program. SIMPL+ provides two constructs for branching the program based on the value of expressions: *if-else* and the *switch-case* statement.

if–else

if-else is the most commonly used branching construct. In its most basic form, it is structured as follows.

```
if (expression1)
{
   // do something here
}
```

Where expression1 represents any valid SIMPL+ expression, including variables, function calls, and operators. If this expression evaluates to TRUE, then the code inside the braces is executed. If this expression evaluates to FALSE, the code inside the braces is skipped.

What is the definition of TRUE and FALSE in SIMPL+? As was discussed in "Working with Data (Variables)" on page 13, expressions, which evaluate to a non-zero result, are considered TRUE, and expressions that evaluate to 0 are considered FALSE. For example, refer to the expressions in the table that follows.

Expressions

EXPRESSION	EVALUATES TO
a = 3	true if a=3, false otherwise
b*4 - a/3	true as long as the result is non-zero
1	always true
0	always false

One limitation with the *if* construct, as shown above, is that the code inside the *if* is run whenever expression1 evaluates as TRUE, but any code after the closing braces runs regardless. It is often useful to execute one set of code when a condition is TRUE and then another set of code if that same condition is FALSE. For this application, use the *if-else* construct, which looks like the following.

```
if (expression1)
{
    // do something if expression1 is true
}
else
{
    // do something else if expression1 is false
}
```

Programming to anticipate user errors and handle them appropriately is called error-trapping. It is a recommended programming practice.

It should be clear that the code following the *if* runs whenever expression1 evaluates to TRUE and the code following the *else* executes whenever expression1 evaluates to FALSE. Obviously, there can never be a case where both sections of code execute together.

The following example is designed to control a CD changer. Before telling the CD player to go to a particular disc number, it checks to see that the analog value, which represents the disc number, does not exceed the maximum value.

```
#DEFINE_CONSTANT NUMDISCS 100

ANALOG_INPUT disc_number;
STRING_OUTPUT CD_command, message;

CHANGE disc_number
{
    if (disc_number <= NUMDISCS)
    {
        CD_command = "DISC " + itoa(disc_number) + "\r";
        message = "Changing to disc " + itoa(disc_number) +
                  "\n";
    }
    else
    {
        message = "Illegal disc number\n";
    }
}
```

There is one last variation on the *if-else* statement. In the example above, the decision to be made is binary. That is, do one thing if this is true, otherwise do something else. In some cases decisions are not that straight forward. For example, to check the current day of the week, execute one set of code if it is Saturday, another set of code if it is Sunday, and yet some other code if it is any other day of the week. One way to accomplish this is by using a series of *if-else* statements. For this example, the code might look like the following.

```
today = getDayNum();       // gets the current day of the week


if (today = 0)             // is today Sunday?
{
   // code to run on Sundays
}
else if (today = 5)        // is today Friday?
{
   // code to run on Friday
}
else if (today = 6)        // is today Saturday?
{
   // code to run on Saturdays
}
else                       // only gets here if the first three
{                          // conditions are false
   // code to run on all other days
}
```

NOTE: There can be as many *if-else* statements in a single construct as necessary. However, sometimes tasks like these are better handled with the *switch - case* construct, discussed in the next section.

Finally, note that *if* statements can be nested inside other *if* statements.

switch–case

In the last section, it was shown that the *if-else* construct can be used for making complex decisions. Also it was used for making a choice between mutually exclusive conditions (conditions that cannot coexist), the syntax can become cumbersome. For this particular case SIMPL+ offers the *switch-case* construct.

Think of *switch-case* as a compact way of writing an *if-else* construct. The basic form of the *switch-case* is shown after this paragraph.

```
switch (expression)
{
   case (expression1):
   {
      // code here executes if
      // expression = expression1
   }
   case (expression2):
   {
      // code here executes if
      // expression = expression2
   }
   default:
   {
      // code here executes if none
      // of the above cases are true
   }
}
```

NOTE: The use of the default keyword allows specific code to execute if none of the other cases are true. This is identical to the final *else* statement in the *if-else* construct mentioned in “if–else”.

Examine an example using the *switch-case* construct. Perhaps there is a variable that should hold the number of days in the current month. The following example uses *switch-case* to set the value of this variable.

```
switch (getMonthNum())
{
    case (2): //February
    {
        if (leapYear) // this variable was set elsewhere
               numdays = 29;
        else
               numdays = 28;
    }
    case (4): // April
        numdays = 30;
    case (6): // June
        numdays = 30;
    case (9): // September
        numdays = 30;
    case (11): // November
        numdays = 30;
    default:  // Any other month
        numdays = 31;
}
```

Notice that curly braces did not enclose many of the statements in the previous example. For most SIMPL+ constructs, the braces are only needed when more than one statement is to be grouped together. If the program has only a single statement following the case keyword, then the braces are optional.

Controlling Program Flow: Loops

"Controlling Program Flow: Branching" (on page 24) discussed constructs for controlling the flow of a program by making decisions and branching. Sometimes a program should execute the same code a number of times. This is called looping. SIMPL+ provides three looping constructs: the *for* loop, the *while* loop, and the *do-until* loop.

for Loops

The *for* loop is useful to cause a section of code to execute a specific number of times. For example, consider clearing each element of a 15-element string array (set it to an empty string). Use a *for* loop set to run 15 times and clear one element each time through the loop.

Control the number of loops a *for* loop executes through the use of an index variable, which must be an integer variable previously declared in the variable declaration section of the program. Specify the starting and ending values for the index variable, and an optional step value (how much the variable increments by each time through the loop). Inside the loop, the executing code can reference this index.

The syntax of the *for* loop is as follows.

```
for (<variable> = <start> to <end> step <stepValue>)
{
    // code in here executes each time through the loop
}
```

To see an example of the *for* loop use the situation alluded to above. That is, a need to clear each string element in a string array. A program to accomplish this might look like the following.

```
DIGITAL_INPUT clearArray;                // a trigger signal
INTEGER i;                               // the index variable
STRING stringArray[50][14];              // a 15-element array

PUSH clearArray                          // event function
{
    for (i = 0 to 14)
    {
        stringArray[i] = "";             // set the ith element
                                         // to an empty string
        print("cleared element %d\n",i); // debug message
    }                                    // this ends the for
                                         // loop
}                                        // this ends the push
                                         // function
```

In this example, the loop index *i* is set to run from 0 to 14, which represents the first and last elements in *stringArray* respectively. Also notice that the step keyword is omitted. This keyword is optional and if it is not used, the loop index increments by 1 each time through the loop. To clear only the even-numbered array elements, the following could have used.

```
    for (i = 0 to 14 step 2)
    { . . . }
```

The step value can also be negative, allowing the loop index to be reduced by some value each time though the loop.

The *for* loop flexibility can be enhanced further by using expressions instead of constant values for the start, end, and step values. For example, there might be a need to add up the value of each byte in a string in order to calculate the value of a checksum character. Since the length of the string can change as the program runs, the number of iterations through the loop is unknown. The following code uses the built-in function, *len*, to determine the length of the string and only run through the *for* loop the necessary number of times. System functions are described in detail in "Using System Functions" on page 30.

```
checksum = 0; // initialize the chucksum variable

/* iterate through the string and add up the bytes.
   Note that the { } braces are not needed here
   because the contents of the for-loop is only
   a single line of code */
for (i = 1 to len(someString))
    checksum = checksum + byte(someString,i);

/* now add the checksum byte on to the string
   using the chr function. Note that in this
   example we only use the low-order byte from
   the checksum variable */
someString = someString + chr(checksum);
```

while and do-until Loops

The *for* loop discussed in an earlier section is useful for iterating through code a specific number of times. However, sometimes the exact number of times a loop should repeat is unknown. Instead, it may be necessary to check to see if a condition is true after each loop to decide whether or not the loop should execute again.

There are two looping constructs in SIMPL+ which allows execution of a loop only for as long as a certain condition is true. These are the *while* and *do-until* loops. The while loop has the following syntax.

```
while (expression)
{
    <statements>
}
```

When the *while* statement is executed, the expression contained in parentheses is evaluated. If this expression evaluates to TRUE, then the statements inside the braces are executed. When the closed brace is reached, the program returns to the *while* statement and reevaluates the expression. At this point the process is repeated. It should become clear that the code inside the braces is executed over and over again as long as the *while* expression remains TRUE.

The nature of the *while* loop means that it is the responsibility of the programmer to ensure that the loop is exited at some point. Unlike the *for* loop discussed previously, this loop does not run for a set number of times and then finishes. Consider the example after this paragraph.

```
x = 5;
while (x < 10)
{
    y = y + x;
    print("Help me out of this loop!\n");
}
```

Endless loops cause the SIMPL+ module (in which they occur) to rerun the same code forever. However, due to the multi-tasking nature of the operating system, an endless loop in one module does not cause the rest of the SIMPL program (including other SIMPL+ modules) to stop running. This is discussed in more detail in "Understanding Processing Order" on page 49.

This example shows an endless loop. That is, a loop that runs forever and never exits. The problem is that the value of the *while* expression never changes once the loop is entered. Thus the expression can never evaluate to FALSE. Include code inside the loop that affects the *while* expression (in this case the variable *x* must be modified in some way) and allows the loop to exit at some point.

The *do-until* looping construct is similar to the *while* loop. The difference lies in where the looping expression is evaluated and how this evaluation affects the loop. To see the difference, examine the form of a *do-until* loop.

```
do
{
    <statements>
} until (expression)
```

From the syntax, it is obvious that the looping expression for a *do-until* appears after the looping code. This expression appears before the code in a *while* loop. This discrepancy affects the way the loop is initially entered. As was shown above, a *while* loop first evaluates the expression to see if it is TRUE. If it is, then the loop runs through one time and then the expression is evaluated again.

The *do-until* loop differs from this in that it always executes at least one time. When the *do* keyword is reached, the code that follows (enclosed in braces) is executed before the value of the *until* expression is evaluated. After the initial pass through

this code, the value of this expression determines whether or not the code should be executed again. Here lies the other difference between the *while* and *do-until*. The *while* loop executes as long as the expression remains TRUE. A *do-until* loop executes until an expression becomes TRUE.

When deciding which type of loop should be used, first understand that using any of the three types of loops discussed here can solve many problems. However, one particular loop is usually better suited for a given application than the others. As a general rule of thumb, when the number of iterations the code should execute is known, a *for* loop is preferred. A *while* or a *do-until* loop is ideal to execute a section of code continuously based on the value of some expression.

Once a *while* or a *do-until* loop is determined suitable for a particular application, the question becomes which one of the two should be used? Once again realize that either one can usually accomplish the goal, but one type of loop may require less coding or be more readable in some cases. The basic question to ask is whether or not the loop needs to run through at least one time. If so, a *do-until* is the best choice. If instead, the value of an expression should be checked, then use the *while* loop.

Exiting from Loops Early

All three loops discussed above have built-in ways to exit. The *for* loop exits when the index variable reaches the stated maximum. The *while* loop exits when the expression becomes FALSE. The *do-until* loop exits when the expression becomes TRUE.

Sometimes programming tasks do not always fall neatly into place regarding loops and it may be desirable (or necessary) to exit a loop prematurely. Consider the following example.

```
INTEGER x,y;

for (x=3 to z)
{
    y = y + x*3 - z*z;
    if (y = 0)
        break;
}
```

Notice that in most (if not all) cases, the need for the break statement could be avoided by the use of a different type of loop. In the example above, this could be accomplished by using a *do-until* loop. Consider the following.

```
x = 3;

do
{
    y = y + x*3 - z*z;
    x = x + 1;
} until ((y = 0) || (x = z))
```

Either technique would be considered acceptable.

Using System Functions

In order to make programming in SIMPL+ simpler and more powerful, the concept of functions is introduced. A function is essentially a more complicated

programming task that has been predefined and given a name. Many of the examples in previous sections of this document have used special types of functions called system functions (or built-in functions). To employ system functions, use the following format.

```
returnValue = FunctionName(Parameter1, Parameter2,...);
```

The above syntax is called a function call, because it tells the function (calls it) to cause / perform an action. Notice that there are three elements to a function call. First, it is identified by *FunctionName*, which must be unique for every system function. The function name itself is followed by a comma-separated parameter list enclosed in parentheses. Each function may have a fixed number of parameters, a variable number of parameters, or no parameters at all. These parameters provide the means to supply information to a function. For example, the *itoa* function converts an integer value into its ASCII equivalent, which is very useful for creating strings to control switchers. The single parameter to this function would be the integer value that needs to be converted. Parameters can be any valid SIMPL+ expression. Consider the statements:

```
returnValue = itoa(154);          // constant param
returnValue = itoa(input_num);    // variable param
returnValue = itoa(x*5-4);        // general expression param
```

All are valid function calls.

The variable to the left of the equal sign in the above function calls is used to hold the return value of the function. This value can be either an integer or a string, depending on the nature of the function. Clearly, if a given function returns an integer value, an integer variable must be used to accept this value and a string variable for those functions that return strings. In the *itoa* examples shown above, the return value is the ASCII string which represents the number being converted. Thus, *returnValue* in this case must be a string variable.

Some functions do not return any values at all. Thus it makes no sense to assign a variable to a function as shown above. In addition, when using functions that do return values, sometimes the return value may not be needed. In both cases, use the *Call* keyword:

```
Call FunctionName(Parameter1, Parameter2, ...);
```

In this case, if the function being called does return a value, it is ignored.

Be aware as well that some functions may need to return more than one value. Since functions can only have at most a single return value, there are some functions that modify the values of the parameters that are passed to it.

SIMPL+ provides a large number of system functions, which are listed under a number of categories in the latest revision of the SIMPL+ Language Reference Guide (Doc. 5797). Many of these system functions are used as examples throughout this manual.

User Defined Functions

The previous section details how to use system functions in SIMPL+. When programming, there may be a need to create functions of your own to simplify your programs. These functions are called user-defined functions. User-defined functions

perform exactly like system functions with the only exception in that they must be defined before they are used.

Function Definitions

Since user-defined functions are created by the user (the SIMPL+ programmer), the user must make sure that the SIMPL+ compiler knows about these functions before the program tries to call them. This is accomplished by creating a function definition, which is different from a function call. Remember from the discussion of system functions that a function call is used to invoke a function. A function definition is what tells the SIMPL+ compiler "this is what this function does".

User functions are used for several reasons. It is not desirable to create one function that performs every task for the entire program. To help better organize program modules, creating several smaller functions makes programming easier to read and understand, and debug. User defined functions can also be called by any other function. Rather than have the same programming logic written out in several functions, one function can be defined with this logic and then called by any other function within the module. This will also greatly reduce the module's size.

To help the reusability of functions, any number of variables can be passed to functions. Variables are passed to functions through the function's argument list. This is also called 'parameter passing', or 'function arguments'. Function arguments can be thought of as passing a value into a function. Other variables or literal values can be passed to function arguments. Function arguments are another way of defining local variables. The difference between declaring a local variable within the function and declaring one as part of the parameter list is that the function argument will have the value of the calling function's variable copied into it.

It is also useful for a function to return a value. A function might be written to compute a value. Another function might want to perform a task and return an error code that can be evaluated by the calling function. Functions can only return at most one value, namely integers or strings. When defining a function, the returning value will determine what type of function to declare. The different types of functions are: FUNCTION, INTEGER_FUNCTION and STRING_FUNCTION. For the 2-series compiler, LONG_FUNCTION, SIGNED_INTEGER and SIGNED_LONG_FUNCTION are also available.

The syntax of a SIMPL+ function call is as follows:

```
FUNCTION MyUserFunction( [parameter1][, parameter2][,
parametern] )
{
    <statements>
}


INTEGER_FUNCTION MyUserIntFunction( [parameter1][,
parameter2][, parametern] )
{
    <statements>
}


STRING_FUNCTION MyUserStrFunction( [parameter1][, parameter2][,
parametern] )
{
    <statements>
}
```

The *FUNCTION* keyword is used to tell the SIMPL+ compiler that what follows is the definition of a function and not a function call. The *FUNCTION* keyword also specifies that there will be no return value for this function. *INTEGER_FUNCTION* and *STRING_FUNCTION* specify that an integer or string value will be returned as the result of the function. These keywords are also called the 'function type'.

The next keyword is a name provided by the SIMPL+ programmer that will become the name for this user function (called the 'function name'). Be sure to use a unique name and not an existing SIMPL+ keyword, system function, or a previously declared variable name. Otherwise, a compile syntax error will result.

Following the function name is the function's argument list. If no arguments are needed within this function, then the list can remain empty. Otherwise, a parameter is defined by giving a variable type and name (i.e., INTEGER myIntArgument). One or more functions are possible by separating each with a comma.

Function definitions are global to the SIMPL+ module in which they are defined. That is, any event function, the Function Main, or even another user-defined function can call a user-defined function that has been defined in the same SIMPL+ module. Functions defined in other modules are not accessible.

When calling a function, it is critical not only that that function has been defined in the SIMPL+ module, but also that this function definition occur before the line of code which calls the function. For example, consider the following.

```
INTEGER x;

PUSH someSignal
{
    call MyUserFunction1();
    x = MyUserFunction2( x, 10 );
}

FUNCTION MyUserFunction1()
{
    print("This is MyFunction1 runnning!\n");
}

INTEGER_FUNCTION MyUserFunction2( INTEGER arg1, STRING arg2 )
{
    print("This is MyFunction2 runnning!\n");
}
```

This code causes a compile error, because the function *MyUserFunction1* has been called before it has been defined. This can easily be remedied by reversing the order:

```
INTEGER x;

FUNCTION MyUserFunction1()
{
   print("This is MyFunction1 runnning!\n");
}

INTEGER_FUNCTION MyUserFunction2( INTEGER arg1, STRING arg2 )
{
   print("This is MyFunction2 runnning!\n");
}

PUSH someSignal
{
   call MyUserFunction1();
   x = MyUserFunction2( x, 10 );
}
```

This program compiles without any problems. Due to this dependence on order, the SIMPL+ module template that appears each time a new program is created provides a place to put function definitions. Notice that this section comes after the input/output and variable definitions, but before the event and main functions. Following the layout suggested by the template should prevent most of these errors.

Defining Local Variables In Functions

The concept of local variables was introduced in the section "All About Variables". In this section we will discuss the topic in greater detail and present a number of examples.

What is a local variable? A local variable is a variable (i.e. an integer or string) with a limited "life span" and limited "scope." You can think of global variables as being immortal. That is, for as long as the control system is plugged in and the program is running, global variables retain their values unless a programming statement modifies them. In addition, global variables can be accessed (either to use their value or to modify them) anywhere in a SIMPL+ program. Here is a simple example:

```
DIGITAL_INPUT go;
INTEGER i,j,k; // define 3 global integers

FUNCTION sillyFunction()
{
  i = j * 2;
  k = i - 1;
}

FUNCTION anotherSillyFunction()
{
  j = i + k;
}

PUSH go
{
  i = 1;
  j = 2;
  k = 3;
```

```
  Print("i = %d, j = %d, k = %d\n", i, j, k);
  Call sillyFunction();
  Print("i = %d, j = %d, k = %d\n", i, j, k);
  Call anotherSillyFunction();
  Print("i = %d, j = %d, k = %d\n", i, j, k);
}
```

In this program, it should be clear to see that both of the functions defined, as well as the push event, have access to the three global integers *i*, *j*, and *k*.

Local variables, on the other hand, can only be accessed within the function in which they are defined. If another user-defined function or event function tries to access a local variable defined elsewhere, a compiler error will be generated. In addition, as soon as the function completes execution, all of the functions' local variables are "destroyed." This means that any value they contained is lost. The next time this function is called the local variables are re-created.

Creating local variables is identical to creating global variables, except that the declaration statement is placed inside of the function definition, as follows:

```
FUNCTION localExample()
{
  INTEGER i, count, test;
  STRING s[100], buf[50];
}
```

Passing Variables to Functions as Arguments

The last section describes how to make your programs easier to read and maintain by defining variables local to functions. However, this does not change the fact that by their very nature most functions require access to one or more variables that have been declared elsewhere in the program, either as global variables or as local variables in a different function. The question is, how can your function access these variables.

As we have already seen, global variables are available everywhere in a program, thus you can simply use such variables to share data between functions and the rest of the program. This is considered bad programming practice, however, and often leads to programs that are hard to read and even harder to debug. Functions can also access any input/output signals defined in the program, but as these signals can be thought of as global variables, this too is not considered good programming practice.

Instead of using global variables to share data, we instead use the concept of passing arguments (also known as parameters) into functions. Arguments can be thought of as an ordered list of variables that are passed to a function by the calling function (the term calling function simply refers to the scope of the code statement which calls the function in question). To define a function's parameters, you list them inside the parentheses following the function name. A typical function definition would look like this:

```
FUNCTION some_function (INTEGER var1, INTEGER var2, STRING
var3)
{
  INTEGER localInt;
  STRING  localStr[100];

  var1 = var1 + 1;
  localInt = var1 + var2;
```

```
  localStr = left(var3, 10);
}
```

Notice that the function shown above has three arguments, named *var1*, *var2*, and *var3*. *var1* and *var2* are integers, while *var3* is a string. Shown below is an example of how to call this function from elsewhere in your program:

```
Call some_function( intVal1, 5+1, stringVal1);
```

Here we are assuming that the variable *int1* has been defined as an integer earlier in the program, as has the string variable, *string1*. Also note that the second argument is a constant, and not a variable at all. This simply means that inside some_function, the value of *var2* will be set to 6.

ByRef, ByVal, and ReadOnlyByRef

When defining a function's argument list, there are optional keywords you can use which give you greater control over the behavior of the arguments. These keywords are *ByRef*, *ByVal*, and *ReadOnlyByRef*.

What do these keywords mean? Essentially they describe the way that SIMPL+ passes variables to the function. When a function argument is defined as *ByRef*, any variable that is passed to this argument will pass enough information about itself to allow the function to modify the value of the original variable. The term *ByRef* is used because we say a "reference" to the original variable is passed to the function. This reference can be thought of the memory location where the original variable lives. When a function argument is defined as "ByVal", only the value of the variable and not the variable itself is passed to the function, so the original variable cannot be modified within the function. As an example, below is a function, which takes two strings as arguments. It inserts one string into the other at a specified character location:

```
FUNCTION insertString(ByRef STRING string1, ByVal STRING
string2, ByVal INTEGER position)
{
  STRING leftpart[20], rightpart[20];

  leftpart = left(string1,position);
  rightpart = right(string1,position);

  string1 = leftpart + string2 + rightpart;
}
```

In this example, note that only the first string argument, *string1*, was defined as *ByRef*.

Functions That Return Values

To this point all user-defined functions we have discussed have had one thing in common: when the functions are finished executing they do not return a value to the calling code. This statement is ambiguous, because some of the functions do modify the values of their arguments, and thus these modified variables can be used by the calling procedure. However, the term return value is used to describe the core value, which is returned from the function to the calling procedure. Many system functions discussed earlier in this manual have return values. For example, here are some statements that use the return values of functions:

```
String1 = itoa(int1);
position = find("Artist",CD_data);
int3 = max(int1, int2);
```

For clarity, here are some example statements using system functions that do not have return values:

```
Print("Variable int1 = %d\n",int1);
ProcessLogic();
CancelWait(VCRWait);
```

It should be clear that, at least as far as system functions go, whether or not a function returns a value depends largely upon what that function is designed to do. For example, the *itoa* function would not be very valuable if it did not return a string, which could then be assigned to a variable, or used inside of an expression. On the other hand, the *Print* function simply outputs text to the console for viewing, and thus no return value is needed.

Allowing a user-defined function to return a value is extremely useful, as it allows such functions to be used in flexible ways, such as inside of an expression. To enable a function to return a value, use a modified version of the function keyword, as shown below:

```
STRING_FUNCTION strfunc1();  //function returns a string
INTEGER_FUNCTION intfunc1(); //function returns an integer
FUNCTION func1();             //function has no return value
```

Clearly, functions defined using the STRING_FUNCTION keyword will return a string value, and those defined using the INTEGER_FUNCTION keyword will return an integer value. Function declared using the function keyword would have no return value.

Once a function has been declared using the appropriate function type, it is the responsibility of the programmer to ensure that the proper value is returned. This is accomplished using the *return* function. To illustrate this, examine the following function example, which raises one number to the power determined by the second argument, and returns the result.

```
INTEGER_FUNCTION power(INTEGER base, INTEGER exponent)
{
  INTEGER i, result;

  if (base = 0)
    return (0);

  else if (exponent = 0)
    return (1);

  else {
    result = 0; // initialize result
    for (i = 1 to exponent)
      result = result + result * base;

    return (result);
  }
}
```

To use this function in a program, simply call the function just like you would any built-in system function. Here are a few usage examples:

```
Print("5 raised to the power of 3 = %d\n",power(5,3));
x = power(y,z);
```

As a second example, we shall build a function which appends a simple checksum byte onto the end of a string. As was mentioned earlier in this manual, checksums are used by many devices to provide a basic form of error checking. In this example, the checksum is formed by simply adding up the values of each byte in the command string and then appending the equivalent ASCII character of the result onto the string itself. If the checksum value is larger than a single byte (255 decimal), we simply ignore the overflow and use the lower 8-bits of the result.

```
DIGITAL_INPUT control_device1, control_device2;
STRING_OUTPUT device1_out, device2_out;
STRING device1_cmd[20], device2_cmd[20], tempCmd[20];

STRING_FUNCTION appendChecksum(STRING command)
{
    INTGEGER checksum, i;  // define local variables

    checksum = 0;    // initialize variable

     for (i = 1 to len(command))   // calculate the sum
        checksum = checksum + byte(command,i);

    return(command + chr(checksum));  //append the byte
}

PUSH vcr_play
{
    vcr_out = appendChecksum("PLAY");
}

PUSH vcr_stop
{
    vcr_out = appendChecksum("STOP");
}
```

In this example, the system function, *byte*, is used inside the function to get the numeric value of each byte in the string. After the checksum has been calculated, the *chr* function is used to append the corresponding ASCII character to the end of the command string. Realize that this example is useful for just one (very simple) type of checksum.

Function Libraries

You are likely to find that the longer you program in SIMPL+ the more you will need to repeat code you have already written. For example, a function that converts the temperature from Celsius to Fahrenheit might come in handy in more than one job.

Clearly, code that has many applications is best placed inside of a function. Remember, however, that unlike system functions, which are globally available, user-defined functions are only available inside of the SIMPL+ program in which they exist. Thus is you need to use a user-defined function in more than one SIMPL+ program, you must copy and paste it from one program to another. While this technique works, it can lead to problems when, for example, you find a bug in the function and fix it one program but forget to change it elsewhere.

To solve this problem, SIMPL+ has introduced the concept of function libraries. Simply put, a function library is a collection of user-defined functions placed in a separate file. A library can consist of only a single function, or can consist of every function you have ever written. More likely, you will organize your libraries so that each one contains related functions. For example, you may create a "string handling" library, which consists of a number of functions that perform useful operations on stings.

Once a function has been included inside of a function library, it now becomes accessible to all SIMPL+ modules that are made aware of it. To make a SIMPL+ program aware of a particular library, you must use the #USER_LIBRARY. To include a user library within a SIMPL+ module, the syntax is as follows:

```
#USER_LIBRARY "MyStringFunctionLib"
```

Note that the file extension (.usl in this case) is left out. The above example refers to the function library called "MyStringFunctionLib.usl." Any number of user libraries can be included within a SIMPL+ module.

Special function libraries that are created by Crestron and made available to all customers can be used in a similar manner. The only difference is the use of the #CRESTRON_LIBRARY compiler directive in place of #USER_LIBRARY. Crestron function library files end with the extension .csl.

Compact Flash Functions

The 2-series control system supports reading and writing to compact flash cards. Certain control systems have a built-in compact flash slot with the ability to easily insert and remove compact flash cards (i.e., AV2, PRO2, and PAC2).

Data storage is a valuable, powerful and important part of programming. The ability to store and retrieve data from a removable data source can provide many useful and powerful solutions. Some of these solutions include the ability to backup data, transferring data from one control system to another, reading and writing data to and from formats that other database programs can recognize, and implementing database-driven programs (the ability for a program to act dynamically based on actions defined in the database).

The SIMPL+ file functions perform file access with the control system's compact flash card. Because of the overhead involved with maintaining current directory and file positions, there are restrictions on file I/O. Each SIMPL+ thread (main loop or event handler) that requires file operations must first identify itself with the operating system. This is done with the function, *StartFileOperations*. Before terminating the thread, the function, *EndFileOperations* must be called. Files cannot be opened across threads. In other words, you cannot open a file in one thread, such as *Function Main*, and then access the file with the returned file handle in another thread, such as an event handler. Files should be opened, accessed and closed within the same thread.

CheckForDisk and WaitForNewDisk

Before accessing compact flash, the program must either first check to see if a compact flash card exists within the control system, or wait for a card to be inserted.

Certain programs might rely on the compact flash card being inserted within the control system. The function, *CheckForDisk*, will test for the existence of a compact

flash card within the control system. The function will return an error code and the program can act accordingly.

Other programs might prompt the end-user to insert a compact flash card. The function, *WaitForNewDisk*, will halt the program and resume when a compact flash card is detected within the control system.

The following is an example of a program that needs to read data from a compact flash card upon startup:

```
FUNCTION ReadMyCompactFlashCard()
{
    // call functions to read the compact flash card
    //
    // Note that this function will exist within the same
    //   thread as the calling function (Function Main).
    //   Because of this, the functions, StartFileOperations
    //   and EndFileOperations should not be used here.
}

Function Main()
{
    StartFileOperations();

    if (CheckForDisk() = 1)
        Call ReadMyCompactFlashCard();
    else if ( WaitForNewDisk() = 0 )
        Call ReadMyCompactFlashCard();

    EndFileOperations();
}
```

If the program is dependent upon data that read in from the compact flash card, it is imperative for the program to validate the existence of the card. Otherwise, the program will not have the necessary data needed to execute properly. The above function will first check if the compact flash card is already inserted into the control system upon system startup. If so, it will call the user-defined function, *ReadMyCompactFlashCard*, to perform any file read operations on the compact flash card. If the compact flash card was not found in the control system, the program will wait for the card to be inserted before continuing. Once inserted, the same function, *ReadMyCompactFlashCard*, is called.

Reading and Writing Data

Once the existence of the compact flash card is verified, the reading and writing of data can be performed. Data can be read or written either with individual elements (i.e., a single integer or string), or with entire structures of data.

Because each datatype (i.e.: INTEGER, STRING, LONG_INTEGER) uses a different amount of storage in memory, there are different functions to read and write each of these types. The return value of each of these functions is the actual number of bytes read or written to the file. The reason why different functions have to be called instead of having just one function is for the following reason. Data elements are written to a file by inserting one element after another. The file does not contain any information as to what that data is or how it is to be extracted out. It is up to the program that will ultimately read that file to know exactly what is contained within the file and how to extract the data back out of it.

The following example demonstrates this:

```
DIGITAL_INPUT readCompactFlashCard;
DIGITAL_INPUT writeCompactFlashCard;

INTEGER myInt;
LONG_INTEGER myLongInt;
STRING myStr[50];

PUSH writeCompactFlashCard
{
    SIGNED_INTEGER nFileHandle;
    INTEGER nNumBytes;

    StartFileOperations()

    nFileHandle = FileOpen( "\\CF0\\MyFile.txt",
                            _O_WRONLY | _O_CREAT );
    if( nFileHandle >= 0 )
    {
        nNumBytes = WriteInteger( nFileHandle, myInt );
        nNumBytes = WriteLongInteger( nFileHandle, myLongInt );
        nNumBytes = WriteString( nFileHandle, myStr );

        FileClose( nFileHandle );
    }

    EndFileOperations();
}

PUSH readCompactFlashCard
{
    SIGNED_INTEGER nFileHandle;
    INTEGER nNumBytes;

    StartFileOperations()

    nFileHandle = FileOpen( "\\CF0\\MyFile.txt", _O_CREAT );
    if( nFileHandle >= 0 )
    {
        nNumBytes = ReadInteger( nFileHandle, myInt );
        nNumBytes = ReadLongInteger( nFileHandle, myLongInt );
        nNumBytes = ReadString( nFileHandle, myStr );

        FileClose( nFileHandle );
    }

    EndFileOperations();
}
```

The functions, *ReadStructure* and *WriteStructure*, automate the reading and writing of the individual fields within the structure. These functions do not return the number of bytes read or written. Instead, both functions have an additional argument that will contain the number of bytes read or written after the function call executes.

The following example demonstrates this:

```
DIGITAL_INPUT readCompactFlashCard;
DIGITAL_INPUT writeCompactFlashCard;

STRUCTURE myStruct
{
    INTEGER myInt;
    LONG_INTEGER myLongInt;
    STRING myStr[50];
```

```
}
myStruct struct;

PUSH writeCompactFlashCard
{
    SIGNED_INTEGER nFileHandle;
    INTEGER nNumBytes;

    StartFileOperations()

    nFileHandle = FileOpen( "\\CF0\\MyFile.txt",
                            _O_WRONLY | _O_CREAT );
    if( nFileHandle >= 0 )
    {
        WriteStructure( nFileHandle, struct, nNumBytes );

        Print( "The number of bytes written = %d", nNumBytes );

        FileClose( nFileHandle );
    }

    EndFileOperations();
}

PUSH readCompactFlashCard
{
    SIGNED_INTEGER nFileHandle;
    INTEGER nNumBytes;

    StartFileOperations()

    nFileHandle = FileOpen( "\\CF0\\MyFile.txt", _O_CREAT );
    if( nFileHandle >= 0 )
    {
        ReadStructure( nFileHandle, myInt, nNumBytes );

        Print( "The number of bytes read = %d", nNumBytes );

        FileClose( nFileHandle );
    }

    EndFileOperations();
}
```

Working with Time

Up until now, this manual has discussed elements of the SIMPL+ language that have no concept of time. This means that each statement in a SIMPL+ program executes after the previous statement is completed. There are times when programming where there is a need to have control over exactly when the statements in the program execute. This section deals with those language constructs.

Delay

The *Delay* function pauses the execution of the current SIMPL+ for the time specified in the parameter field. As with most time values in SIMPL+, this time value is specified in hundredths of seconds. The following program causes the program to stop for five seconds before resuming.

```
PUSH startMe
{
    Print("I'm starting now...");
    Delay(500);      //this equals 5 seconds
    Print("and I'm ending 5 seconds later.\n");
}
```

It is important to realize that the control system never allows a SIMPL+ program to "lock up" the rest of the system for any significant amount of time. Thus, whenever a delay function is reached, the control system performs a task switch, meaning that the execution of the current SIMPL+ module stops momentarily as the natural flow through the SIMPL program continues. In this case, even though the SIMPL+ program has stopped for five seconds, other code in the overall SIMPL program (including standard logic symbols and other SIMPL+ modules) continues to operate normally. The concept of task switching is covered in more detail in the section marked "Understanding Processing Order" on page 49.

Pulse

The *Pulse* function is used to drive a digital output signal high for a specified amount of time. Once again, the time is specified in hundredths of seconds. When a *Pulse* statement is executed, the digital output signal specified is driven high and a task switch occurs. Such a task switch is necessary in order for the rest of the SIMPL program to recognize that the digital signal has indeed gone high. After the time specified has expired, the digital output is driven low and another task switch occurs. The following program causes the digital output signal, *preset_1,* to be pulsed for a half a second.

```
#DEFINE_CONSTANT PULSE_TIME 50

DIGITAL_OUTPUT preset_1, preset_2, preset_3;


PUSH some_input
{
    Pulse(PULSE_TIME, preset_1);
}
```

The *Pulse* function is very similar in operation to the SIMPL One Shot symbol. In fact, in many cases, it may be more convenient (or more sensible) to simply connect a One Shot, or Multiple One Shot symbols to the output signals of a SIMPL+ module.

Also notice that unlike the *Delay* function, *Pulse* does not cause a pause in the execution of the SIMPL+ code. Therefore, the statements that follow the *Pulse* execute immediately and do not wait for the expiration of the pulse time.

Wait Events

Wait events in SIMPL+ allow operations that are somewhat similar to the *Delay* SIMPL logic symbol. The syntax for a wait event is as follows.

```
Wait (wait_time [, wait_name])
{
    <statements>
}
```

This syntax defines a *wait* event to occur at some time in the future, defined by the value of wait_time. While the wait event is pending execution, it is said to have been "scheduled". The wait event may have an optional name, which can be used to refer back to the event elsewhere in the code.

When a *wait* event definition is reached during execution, the execution of the statements inside the braces (these braces are not needed if the event is only one statement long) is deferred until the time defined by wait_time has expired. Until this occurs, the remainder of the SIMPL+ program executes. If a *wait* event definition is nested inside of a loop, it is possible that it can be reached multiple times before it even executes once. If a *wait* event is pending (i.e., has been scheduled, but not executed), it is not scheduled again until it has been completed.

Once a *wait* event has been scheduled to execute at some later point in time, various operations can be performed on the event before it actually executes. However, only named *wait* events can be modified in this manner, since it is necessary to use the name to refer to the event. The table on the next page lists the available functions, which can operate on *wait* events.

Functions Available During Wait Events

FUNCTION	DESCRIPTION
CancelWait(name)	Removes the named wait from the schedule. The code never executes.
CancelAllWait()	Removes all pending waits from the schedule.
PauseWait(name)	Stops the timer for the named wait. The code does not execute until the timer is started again using ResumeWait().
ResumeWait(name)	Resumes the timer for the named wait, which had been paused earlier.
PauseAllWait()	Similar to PauseWait(), but acts on all pending wait events.
ResumeAllWait()	Similar to ResumeWait(), but acts on all paused wait events.
RetimeWait(time, name)	Sets the time for a pending wait event to the value specified.

This example shows a typical use of wait events. Here, the **SYSTEM ON** button starts a power up sequence and the **SYSTEM OFF** button likewise starts a power down sequence.

```
#DEFINE_CONSTANT   PULSETIME   50   // half second

DIGITAL_INPUT system_on, system_off;
DIGITAL_OUTPUT screen_up, screen_down, lift_up, lift_down;
DIGITAL_OUTPUT vcr_on, vcr_off, dvd_on, dvd_off;
DIGITAL_OUTPUT vproj_on, vproj_off;
DIGITAL_OUTPUT vproj_video1, vproj_video2, vproj_rgb;
DIGITAL_OUTPUT lights_pre_1, lights_pre_2, lights_pre_3;


PUSH system_on
{
    CancelWait(sysOffWait); // cancel the system off wait event


    Pulse(2000, screen_down); // lower screen for 20 sec.
    Pulse(9500, lift_down);   // lower lift for 9.5 sec.


    Wait (1000, sysOnWait1) // 10 second delay
    {
        Pulse(PULSETIME, vcr_on);
        Pulse(PULSETIME, dvd_on);
        Pulse(PULSETIME, lights_pre_1);
```

```
        Pulse(PULSETIME, vproj_on);
    }


    Wait (1500, sysOnWait2) // 15 second delay
        pulse(PULSETIME, vproj_video);
} // end of push event


PUSH system_off
{
    CancelWait(sysOnWait1);
    CancelWait(sysOnWait2);


    Pulse(2000, screen_up);
    Pulse(PULSETIME, vproj_off);
    Pulse(PULSETIME, vcr_off);
    Pulse(PULSETIME, dvd_off);


    Wait(500, sysOffWait)
    {
        Pulse(9500, lift_up);
        Pulse(PULSETIME, lights_pre_3);
    }
} // end of push event
```

Notice that in this example, the *CancelWait* function was used to cancel any pending waits when the **SYSTEM ON** or **SYSTEM OFF** buttons were pressed. This is analogous to using the reset input on the Delay symbol in SIMPL.

Working with Strings

In "Working with Data (Variables)" on page 13 the concept of the BUFFER_INPUT was discussed. This section provides a more in-depth treatment of working with incoming serial data.

BUFFER_INPUT

To review what was discussed earlier, serial data entering a SIMPL+ program may be treated as either a STRING_INPUT or as a BUFFER_INPUT. What is the difference, and which one should be used?

The difference between a STRING_INPUT and BUFFER_INPUT is quite simple. The value of a STRING_INPUT is always the last value of the serial signal that feeds it from the SIMPL program. This means that every time new data is generated on the serial signal in the SIMPL program, the STRING_INPUT variable in the SIMPL+ program changes to contain that data; any data that was previously contained in that variable is lost.

BUFFER_INPUTs on the other hand do not lose any data that was stored there previously. Instead, any new data that is generated onto the serial signal in the SIMPL program is appended to the data currently in the BUFFER_INPUT variable.

The Serial Send symbol simply generates the static text defined in its parameter field onto the output serial signal whenever the trigger input sees a rising signal.

To make this concept even clearer, consider the following simple example. The SIMPL program shown below contains two Serial Send symbols, each one triggered by a button press. The outputs of these symbols are tied together so that both symbols can generate a string onto the same serial signal. Next this signal is connected in two places to the SIMPL+ module. The first input is mapped to a STRING_INPUT and the second is mapped to a BUFFER_INPUT. The declaration section for this module should appear as follows.

```
STRING_INPUT theString[100];
BUFFER_INPUT theBuffer[100];
```

The table below shows the state of these two input variables in response to button presses.

States of Two Input Variables

ACTION	theString	theBuffer
system initializes	empty	empty
button 1 pressed	“Now is”	“Now is”
button 2 pressed	“the time”	“Now is the time”
button 1 pressed	“Now is”	“Now is the time Now is”

From this table, notice that each time the serial signal changes, *theString* assumes this value and the old data stored there is lost. On the other hand, *theBuffer* retains any old data and simply appends the new data onto the end.

A logic wave is the time needed for a signal to propagate from the input to the output of a single logic symbol. This concept is discussed fully in “Understanding Processing Order” on page 49.

Each application should dictate whether it is appropriate to use a STRING_INPUT or a BUFFER_INPUT. In general, use STRING_INPUTs when the serial signal that is feeding it is being driven from a logic symbol like a Serial Send, Analog to Serial, or Serial Gather. In these cases, the serial data is issued on a single logic wave. Therefore, it is certain that the entire string is copied into the STRING_INPUT.

If, on the other hand, the signal feeding into the SIMPL+ program comes from a streaming source such as a serial port, use a BUFFER_INPUT, which can gather up the data as it “dribbles in.”

To solidify this concept, consider another example. Say the program is written for a CD jukebox, which is capable of sending text strings containing the current song information. Typical data received from this device might appear as the following.

```
Artist=Frank Sinatra, Track=My Way, Album=Very Good Years<CR>
```

Where the <CR> at the end of the string represents a carriage return character. This is a relatively long string of data and it is quite possible, even probable, that the operating system would not remove the entire string from the serial port in one piece. This is due to the fact that the control system checks the state of the serial ports very often and removes any data that is found there. Since this data takes some time to reach the port (depending on the baud rate), it is likely that the port’s input buffer is collected before the whole string is there. If there was a serial signal called *jukebox_in* connected to the rx terminal on the COM port definition, the program might be written as follows.

```
first pass:

    jukebox_in = "Artist=Frank Sinatra, Trac"
```

```
second pass:

    jukebox_in = "k=My Way, Album=Very Good Yea"

third pass:

    jukebox_in = "rs<CR>"
```

If this signal, *jukebox_in*, were then connected to a STRING_INPUT of a SIMPL+ program, it is likely that the string might not be seen as one complete piece. Thus the artist's name, the track name, and the album name might not be parsed out for display on a touchpanel. On the other hand, if a BUFFER_INPUT were used instead, this buffer would collect the data as it arrived. Therefore, after the processor read the port the third time, this BUFFER_INPUT would contain the complete string.

Removing Data From Buffers

Once data has been routed into a BUFFER_INPUT, techniques are required to extract data from it. Typically the first thing to be done with data on a BUFFER_INPUT is to pull off a completed command and store it into a separate variable. For example, most data that comes from other devices are delimited with a certain character (or characters) to denote the end of the command. In many instances a carriage return (or carriage return followed by a line feed) is used.

The *getc* function is the most basic way to remove data from a buffer. Each call of *getc* pulls one character out of the buffer and returns that character's ASCII value as the function's return value. Characters are removed from the buffer in the order they arrived. Thus the first character in becomes the first character out. This function now provides the ability to extract data until the desired delimiter is seen. For example, the following code is read data from the buffer until a carriage return is seen.

```
BUFFER_INPUT data_in[100];
INTEGER nextChar;
STRING temp[50], line[50];


CHANGE data_in // trigger whenever a character comes in
{
    do
    {
        nextChar = getc(data_in); // get the next character
        temp = temp + chr(nextChar);
        if (nextChar = 0x0D) // is it a carriage return?
        {
               line = temp;
               temp = "";
        }
    } until (len(data_in) = 0) // empty the buffer
}


Function Main()
{
    temp = "";
}
```

Notice that a *do-until* loop was used in the example above. Every time a change event is triggered for the *data_in* buffer, it is uncertain that only one character has been inserted. In fact, many characters may have been added since the last change event. Due to this possibility, continue to pull characters out of the buffer with *getc* until the buffer is empty, which is what the expression (len(data_in) = 0) reveals.

Also notice from the example that the extracted character is stored into an integer. This is because *getc* returns the ASCII value of the character, which is an integer. On the next line, the *chr* function is used to convert that value into a one-byte string, which can be added to temp.

Although this example should work for real-world applications, there is a potential problem should multiple lines of data come in on the same logic wave. Should this happen, only the last complete line is stored into *line* and the rest is lost. To account for this, make *line* into a string array and store each subsequent line into a different array element. Another possibility is that any code that is needed to further act upon the data could be built directly into this loop. Thus removing the need to store more than one line of data.

Once the data has been removed from the buffer and stored in a known format (in this case, one complete command from the device), the desired data can be extracted. Using the example above where the data was coming from a CD jukebox, the following example could be used to extract the artist, track, and album title.

```
BUFFER_INPUT jukebox[100];
STRING_OUTPUT artist, track, album;
INTEGER startPos;
STRING searchStr[20], tempStr[100];


CHANGE jukebox
{
  do
  {
    tempStr = tempStr + chr(getc(jukebox));
    if ( right(tempStr,1) = "\r" )
    {
      searchStr = "Artist=";
      startPos = Find(searchStr,tempStr);
      if (startPos) {  // was the string found?
        startPos = startPos + len(searchStr);
        artist = mid(tempStr, startpos,
                  Find(",",tempStr,startpos) - startpos);
        searchStr = "Track=";
        startpos = Find(searchStr,tempStr) + len(searchStr);
        track = mid(tempStr, startpos,
                 Find("\r",tempStr,startpos) - startpos);
        searchStr = "Album=";
        startpos = Find(searchStr,tempStr) + len(searchStr);
        album = mid(tempStr, startpos,
                  Find("\r",tempStr,startpos) - startpos);
        tempStr = "";
      }
    }


  } until (len(jukebox) = 0);
}


Function Main()
{
```

```
    tempStr = "";
}
```

This example introduces two new system functions, which are extremely useful for string manipulation, the *Find* and *Mid* functions. To search for the existence of a substring inside of another string, use *Find*. If it is located, the return value of the function is the character position where this string was found. If the substring was not found, then *Find* returns zero. Notice that towards the top of the example the program checked to see if the substring "Artist=" is found in the string. If it is not, then assume that the incoming data was of another format and there is no need to bother looking for the other search strings ("Track=" and "Album=").

Understanding Processing Order

How SIMPL+ and SIMPL Interact

Advanced SIMPL programmers should be familiar with how logic is processed in SIMPL programs. This guide does not attempt to explain this concept here, but it does detail how SIMPL+ programs fit into the picture. However, a couple of definitions may be helpful.

Logic Wave - The act of propagating signals from the input to the output of a logic symbol. In a typical program, a single logic wave may include the processing of many symbols.

Logic Solution - An arbitrary number of logic waves, processed until the state of all signals in the program have settled to a stable (i.e. unchanging) state.

In general, when a SIMPL+ event function is triggered, it is processed to conclusion in a single logic wave. That is, if this event caused a change to one of the output signals, that signal would be valid one logic wave after the event was triggered. In this simple case, a SIMPL+ program acts identically to a SIMPL logic symbol from a timing standpoint. In addition, for multiple SIMPL+ events triggered on the same logic wave (whether or not they are in the same module), these events multi-task (run at the same time) and complete before the next wave.

As SIMPL+ programs become more complex and processor intensive however, this general rule may no longer apply. Instead, the operating system may determine that too much time has elapsed and temporarily suspend the SIMPL+ program while it continues to process the SIMPL logic (which may also include other SIMPL+ programs). For example, if an event function must run through a loop 2000 times before completing, the processor may decide to perform a task switch and process other logic before completing the loop.

This task-switching ability has the benefit of not freezing up the rest of the program while a particularly intensive calculation is proceeding. After the completion of the current logic solution, the SIMPL+ program that was exited continues from where it left off.

Forcing a Task Switch

There may be times in programming when it is necessary to force a task switch to occur. For example, when a digital_output is set high, it normally is not propagated to the SIMPL program until the SIMPL+ completes. To guarantee that the digital signal is asserted, force the system to switch from the SIMPL+ module back into the SIMPL program.

There are two ways to force a task switch: with the ProcessLogic function or the Delay function. To provide an immediate task switch out of the current SIMPL+ module use ProcessLogic. When the logic processor enters this module on the next logic solution, execution begins with the line immediately following. An immediate task switch also results from Delay, but the SIMPL+ module does not continue executing until the time specified has elapsed. The Delay function is discussed in greater detail in "Working with Time" on page 42.

Debugging

Rare is the case where a program works perfectly on the first try. Usually, a programmer must make numerous modifications to the original code in order to get the desired results. Programming bugs can be mistakes in syntax, typos, design errors, or a misunderstanding of certain language elements. This section is not intended to prevent mistakes, but rather to find and fix them.

Compiler Errors

Of all the possible problems that a program can have, ones that cause the compiler to complain are perhaps the easiest to remedy. The reason is quite simple: the compiler reveals what the problem is and where in the program it is located. The only job is to recognize exactly what the compiler means and make the necessary changes.

The following list provides the most common causes of compiler errors.

- Missing a semi-colon at the end of a statement
- Having a semi-colon where it does not belong (e.g., before an opening brace of a compound statement)
- Trying to use a variable that has not been declared, or misspelling a variable
- Attempting to assign a value to an input variable (digital, analog, string, or buffer)
- Syntax errors

If multiple error messages are received when compiling the program, always work with the first one before attempting to fix the rest. Many times, a missing semi-colon at the beginning of a program can confuse the compiler enough that it thinks there are many more errors. Fixing the first error may clear up the rest.

Run-time Errors

The term run-time error refers to an error, which is not caught by the compiler, but causes the program to crash while it is running. For example, consider the following statement.

```
x = y / z;
```

The compiler passes by this statement in the program with no problem, as it should. This is a perfectly valid statement. However, if during run-time, the variable z contains 0 when this statement executes, this becomes an illegal operation. Although the control system is robust enough to catch most errors like this and the system should not crash, unexpected results may occur.

To determine if a run-time error is occurring in your program, watch the status of the control system's computer port with a program such as the Viewport (available through SIMPL Windows or Crestron VisionTools® Pro-e). An "Interrupt" message or some other error message can clue the user in to a problem. Locate the problem by strategically placing *Print* statements in the code. The *Print* statement is covered in the next section.

Debugging with Print()

The most powerful debugging tool for use with SIMPL+ is the *Print* function. This function allows the user to print out messages and variable contents during program execution. The data that is generated by the *Print* function is sent to the computer port of the control system, making it viewable using a terminal emulation program such as the Viewport tool that comes with SIMPL Windows and VisionTools Pro.

The *Print* function is nearly identical to the *MakeString* function discussed in "Working with Data (Variables)". The only difference between the two functions is that *MakeString* generates a formatted string into a string variable, while *Print* generates a formatted string and spits it out of the control system's computer port.

The syntax of *Print* is provided in the following example.

```
Print("<specification string>",<variable list>);
```

The <specification string> is a string, which determines what the output looks like. It can contain static text intermixed with format specifiers. A format specifier is a percentage sign (%) followed by a type character. For example, %d is a format specifier for a decimal value, and %s is the format specifier for a string. Consider this specific example.

```
INTEGER extension;
STRING name[30];

PUSH print_me
{
   extension = 275;
   name = "Joe Cool";
   Print("%s is at extension %d", name, extension);
}
```

When this program is run and the digital input, *print_me,* goes high, the following text is output from the computer port:

Joe Cool is at extension 275

The *Print* statement works by replacing the format specifiers in the specification string with the value of the variables in the variable list. Notice that the order of the format specifiers must match the order of the variables in the list. In this example, the first format specifier encountered is %s, which corresponds to the string name. The next specifier is %d, which corresponds to the integer extension. If the variables in the list were reversed and the specification string kept the same, the output would be unpredictable because the system would try to print extension as a string and name as an integer.

Refer to the latest revision of the SIMPL+ Language Reference Guide (Doc. 5797) for a complete listing of all available format specifiers for use with the *Print* and *MakeString*.

Software License Agreement

This License Agreement ("Agreement") is a legal contract between you (either an individual or a single business entity) and Crestron Electronics, Inc. ("Crestron") for software referenced in this guide, which includes computer software and, as applicable, associated media, printed materials, and "online" or electronic documentation (the "Software").

BY INSTALLING, COPYING, OR OTHERWISE USING THE SOFTWARE, YOU REPRESENT THAT YOU ARE AN AUTHORIZED DEALER OF CRESTRON PRODUCTS OR A CRESTRON AUTHORIZED INDEPENDENT PROGRAMMER AND YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT INSTALL OR USE THE SOFTWARE.

IF YOU HAVE PAID A FEE FOR THIS LICENSE AND DO NOT ACCEPT THE TERMS OF THIS AGREEMENT, CRESTRON WILL REFUND THE FEE TO YOU PROVIDED YOU (1) CLICK THE DO NOT ACCEPT BUTTON, (2) DO NOT INSTALL THE SOFTWARE AND (3) RETURN ALL SOFTWARE, MEDIA AND OTHER DOCUMENTATION AND MATERIALS PROVIDED WITH THE SOFTWARE TO CRESTRON AT: CRESTRON ELECTRONICS, INC., 15 VOLVO DRIVE, ROCKLEIGH, NEW JERSEY 07647, WITHIN 30 DAYS OF PAYMENT.

LICENSE TERMS

Crestron hereby grants You and You accept a nonexclusive, nontransferable license to use the Software (a) in machine readable object code together with the related explanatory written materials provided by Creston (b) on a central processing unit ("CPU") owned or leased or otherwise controlled exclusively by You, and (c) only as authorized in this Agreement and the related explanatory files and written materials provided by Crestron.

If this software requires payment for a license, you may make one backup copy of the Software, provided Yo*u*r backup copy is not installed or used on any CPU. You may not transfer the rights of this Agreement to a backup copy unless the installed copy of the Software is destroyed or otherwise inoperable and You transfer all rights in the Software.

You may not transfer the license granted pursuant to this Agreement or assign this Agreement without the express written consent of Crestron.

If this software requires payment for a license, the total number of CPU's on which all versions of the Software are installed may not exceed one per license fee (1) and no concurrent, server or network use of the Software (including any permitted back-up copies) is permitted, including but not limited to using the Software (a) either directly or through commands, data or instructions from or to another computer (b) for local, campus or wide area network, internet or web hosting services; or (c) pursuant to any rental, sharing or "service bureau" arrangement.

The Software is designed as a software development and customization tool. As such Crestron cannot and does not guarantee any results of use of the Software or that the Software will operate error free and You acknowledge that any development that You perform using the Software or Host Application is done entirely at Your own risk.

The Software is licensed and not sold. Crestron retains ownership of the Software and all copies of the Software and reserves all rights not expressly granted in writing.

OTHER LIMITATIONS

You must be an Authorized Dealer of Crestron products or a Crestron Authorized Independent Programmer to install or use the Software. If Your status as a Crestron Authorized Dealer or Crestron Authorized Independent Programmer is terminated, Your license is also terminated.

You may not rent, lease, lend, sublicense, distribute or otherwise transfer or assign any interest in or to the Software.

You may not reverse engineer, decompile, or disassemble the Software.

You agree that the Software will not be shipped, transferred or exported into any country or used in any manner prohibited by the United States Export Administration Act or any other export laws, restrictions or regulations ("Export Laws"). By downloading or installing the Software You (a) are certifying that You are not a national of Cuba, Iran, Iraq, Libya, North Korea, Sudan, or Syria or any country to which the United States embargoes goods (b) are certifying that You are not otherwise prohibited from receiving the Software and (c) You agree to comply with the Export Laws.

If any part of this Agreement is found void and unenforceable, it will not affect the validity of the balance of the Agreement, which shall remain valid and enforceable according to its terms. This Agreement may only be modified by a writing signed by an authorized officer of Crestron. Updates may be licensed to You by Crestron with additional or different terms. This is the entire agreement between Crestron and You relating to the Software and it supersedes any prior representations, discussions, undertakings, communications or advertising relating to the Software. The failure of either party to enforce any right or take any action in the event of a breach hereunder shall constitute a waiver unless expressly acknowledged and set forth in writing by the party alleged to have provided such waiver.

If You are a business or organization, You agree that upon request from Crestron or its authorized agent, You will within thirty (30) days fully document and certify that use of any and all Software at the time of the request is in conformity with Your valid licenses from Crestron of its authorized agent.

Without prejudice to any other rights, Crestron may terminate this Agreement immediately upon notice if you fail to comply with the terms and conditions of this Agreement. In such event, you must destroy all copies of the Software and all of its component parts.

PROPRIETARY RIGHTS

Copyright. All title and copyrights in and to the Software (including, without limitation, any images, photographs, animations, video, audio, music, text, and "applets" incorporated into the Software), the accompanying media and printed materials, and any copies of the Software are owned by Crestron or its suppliers. The Software is protected by copyright laws and international treaty provisions. Therefore, you must treat the Software like any other copyrighted material, subject to the provisions of this Agreement.

Submissions. Should you decide to transmit to Crestron's website by any means or by any media any materials or other information (including, without limitation, ideas, concepts or techniques for new or improved services and products), whether as information, feedback, data, questions, comments, suggestions or the like, you agree such submissions are unrestricted and shall be deemed non-confidential and you automatically grant Crestron and its assigns a non-exclusive, royalty-tree, worldwide, perpetual, irrevocable license, with the right to sublicense, to use, copy, transmit, distribute, create derivative works of, display and perform the same.

Trademarks. CRESTRON and the Swirl Logo are registered trademarks of Crestron Electronics, Inc. You shall not remove or conceal any trademark or proprietary notice of Crestron from the Software including any back-up copy.

GOVERNING LAW

This Agreement shall be governed by the laws of the State of New Jersey, without regard to conflicts of laws principles. Any disputes between the parties to the Agreement shall be brought in the state courts in Bergen County, New Jersey or the federal courts located in the District of New Jersey. The United Nations Convention on Contracts for the International Sale of Goods, shall not apply to this Agreement.

CRESTRON LIMITED WARRANTY

CRESTRON warrants that: (a) the Software will perform substantially in accordance with the published specifications for a period of ninety (90) days from the date of receipt, and (b) that any hardware accompanying the Software will be subject to its own limited warranty as stated in its accompanying written material. Crestron shall, at its option, repair or replace or refund the license fee for any Software found defective by Crestron if notified by you within the warranty period. The foregoing remedy shall be your exclusive remedy for any claim or loss arising from the Software.

CRESTRON shall not be liable to honor warranty terms if the product has been used in any application other than that for which it was intended, or if it as been subjected to misuse, accidental damage, modification, or improper installation procedures. Furthermore, this warranty does not cover any product that has had the serial number or license code altered, defaced, improperly obtained, or removed.

Notwithstanding any agreement to maintain or correct errors or defects Crestron, shall have no obligation to service or correct any error or defect that is not reproducible by Crestron or is deemed in Crestron's reasonable discretion to have resulted from (1) accident; unusual stress; neglect; misuse; failure of electric power, operation of the Software with other media not meeting or not maintained in accordance with the manufacturer's specifications; or causes other than ordinary use; (2) improper installation by anyone other than Crestron or its authorized agents of the Software that deviates from any operating procedures established by Crestron in the material and files provided to You by Crestron or its authorized agent; (3) use of the Software on unauthorized hardware; or (4) modification of, alteration of, or additions to the Software undertaken by persons other than Crestron or Crestron's authorized agents.

ANY LIABILITY OF CRESTRON FOR A DEFECTIVE COPY OF THE SOFTWARE WILL BE LIMITED EXCLUSIVELY TO REPAIR OR REPLACEMENT OF YOUR COPY OF THE SOFTWARE WITH ANOTHER COPY OR REFUND OF THE INITIAL LICENSE FEE CRESTRON RECEIVED FROM YOU FOR THE DEFECTIVE COPY OF THE PRODUCT. THIS WARRANTY SHALL BE THE SOLE AND EXCLUSIVE REMEDY TO YOU. IN NO EVENT SHALL CRESTRON BE LIABLE FOR INCIDENTAL, CONSEQUENTIAL, SPECIAL OR PUNITIVE DAMAGES OF ANY KIND (PROPERTY OR ECONOMIC DAMAGES INCLUSIVE), EVEN IF A CRESTRON REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR OF ANY CLAIM BY ANY THIRD PARTY. CRESTRON MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO TITLE OR INFRINGEMENT OF THIRD-PARTY RIGHTS, MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY OTHER WARRANTIES, NOR AUTHORIZES ANY OTHER PARTY TO OFFER ANY WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY FOR THIS PRODUCT. THIS WARRANTY STATEMENT SUPERSEDES ALL PREVIOUS WARRANTIES.

Return and Warranty Policies

Merchandise Returns / Repair Service

1. No merchandise may be returned for credit, exchange, or service without prior authorization from CRESTRON. To obtain warranty service for CRESTRON products, contact the factory and request an RMA (Return Merchandise Authorization) number. Enclose a note specifying the nature of the problem, name and phone number of contact person, RMA number, and return address.
2. Products may be returned for credit, exchange, or service with a CRESTRON Return Merchandise Authorization (RMA) number. Authorized returns must be shipped freight prepaid to CRESTRON, Cresskill, N.J., or its authorized subsidiaries, with RMA number clearly marked on the outside of all cartons. Shipments arriving freight collect or without an RMA number shall be subject to refusal. CRESTRON reserves the right in its sole and absolute discretion to charge a 15% restocking fee, plus shipping costs, on any products returned with an RMA.
3. Return freight charges following repair of items under warranty shall be paid by CRESTRON, shipping by standard ground carrier. In the event repairs are found to be non-warranty, return freight costs shall be paid by the purchaser.

CRESTRON Limited Warranty

CRESTRON ELECTRONICS, Inc. warrants its products to be free from manufacturing defects in materials and workmanship under normal use for a period of three (3) years from the date of purchase from CRESTRON, with the following exceptions: disk drives and any other moving or rotating mechanical parts, pan/tilt heads and power supplies are covered for a period of one (1) year; touchscreen display and overlay components are covered for 90 days; batteries and incandescent lamps are not covered.

This warranty extends to products purchased directly from CRESTRON or an authorized CRESTRON dealer. Purchasers should inquire of the dealer regarding the nature and extent of the dealer's warranty, if any.

CRESTRON shall not be liable to honor the terms of this warranty if the product has been used in any application other than that for which it was intended, or if it has been subjected to misuse, accidental damage, modification, or improper installation procedures. Furthermore, this warranty does not cover any product that has had the serial number altered, defaced, or removed.

This warranty shall be the sole and exclusive remedy to the original purchaser. In no event shall CRESTRON be liable for incidental or consequential damages of any kind (property or economic damages inclusive) arising from the sale or use of this equipment. CRESTRON is not liable for any claim made by a third party or made by the purchaser for a third party.

CRESTRON shall, at its option, repair or replace any product found defective, without charge for parts or labor. Repaired or replaced equipment and parts supplied under this warranty shall be covered only by the unexpired portion of the warranty.

Except as expressly set forth in this warranty, CRESTRON makes no other warranties, expressed or implied, nor authorizes any other party to offer any warranty, including any implied warranties of merchantability or fitness for a particular purpose. Any implied warranties that may be imposed by law are limited to the terms of this limited warranty. This warranty statement supercedes all previous warranties.

This page intentionally left blank.

Crestron Electronics, Inc.
15 Volvo Drive Rockleigh, NJ 07647
Tel: 888.CRESTRON
Fax: 201.767.7576
www.crestron.com

Programming Guide – DOC. 5789A
04.03